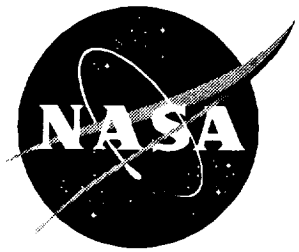
NASA/TM-1998-207666



Framework for Small-Scale Experiments in Software Engineering

Guidance and Control Software Project: Software Engineering Case Study

Kelly J. Hayhurst
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

May 1998

Acknowledgments

The Guidance and Control Software (GCS) project was started in 1985 and has gone through many significant changes since that time. Earle Migneault initiated the project, and the Research Triangle Institute participated in the early phases. In 1988, George Finelli established a valuable connection with the Federal Aviation Administration (FAA); this connection remains in place today. In recent years, Bernice Becher, Andy Boney, Philip Morris, Patrick Quach, Laura Smith, and Debbie Taylor have worked very hard to complete the development component of this project. My thanks go to all these people.

Available from the following:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 487-4650

Contents

Acronyms

AAS	Advanced Automation System
AC	Advisory Circular
ACT	Analysis of Complexity Tool
AR	action report
ARSP	Altimeter Radar Sensor Processing
BCS	Boeing Computer Services
CASE	computer-aided software engineering
CMS	Code Management System
CP	Communications Processing
FAA	Federal Aviation Administration
FAR	Federal Aviation Regulations
GCS	Guidance and Control Software
ID	Identification
LaRC	Langley Research Center
MC/DC	modified condition/decision coverage
PR	problem report
PSAC	Plan for Software Aspects of Certification
RTI	Research Triangle Institute
SDCR	Support Documentation Change Report
TDLRSP	Touchdown Landing Radar Sensor Processing
TSP	Temperature Sensor Processing

Abstract

Software is becoming increasingly significant in today's critical avionics systems. To achieve safe, reliable software, government regulatory agencies such as the Federal Aviation Administration (FAA) and the Department of Defense mandate the use of certain software development methods. However, little scientific evidence exists to show a correlation between software development methods and product quality. Given this lack of evidence, a series of experiments has been conducted to understand why and how software fails. The Guidance and Control Software (GCS) project is the latest in this series. The GCS project is a case study of the Requirements and Technical Concepts for Aviation RTCA/DO-178B guidelines, Software Considerations in Airborne Systems and Equipment Certification. All civil transport airframe and equipment vendors are expected to comply with these guidelines in building systems to be certified by the FAA for use in commercial aircraft. For the case study, two implementations of a guidance and control application were developed to comply with the DO-178B guidelines for Level A (critical) software. The development included the requirements, design, coding, verification, configuration management, and quality assurance processes. This paper discusses the details of the GCS project and presents the results of the case study.

1. Introduction

The replacement of individual gauges and boxes with electronic flight instrument systems began in the late 1970's and the mid-1980's on such aircraft as the Boeing 757/767 and 737-300/400, the Airbus A310, and the McDonnell Douglas MD-80. The potential effects of these electronic flight instrument systems have been compared with those of the jet engine (ref. 1). Software has accompanied the use of electronic systems and has become increasingly important in today's avionics systems, especially in critical tasks that require a high level of reliability and safety. According to Mellor (ref. 2), the amount of software used in modern commercial transport aircraft doubles approximately every 2 years. For example, the Airbus A310 has approximately 5 megabytes, the A320 has approximately 10 megabytes, and the A340 has approximately 20 megabytes of software on board (ref. 1).

As an example of the crucial role that software plays in the aircraft industry today, consider the July 2, 1994, crash of a USAir DC-9 attempting to land at Charlotte, North Carolina, which killed 37 people. As a result of the investigation of that crash, the Federal Aviation Administration (FAA) determined that a software design feature in the wind-shear detection system delayed the detection of wind shear when the wing flaps of the aircraft were in transition. The FAA issued Airworthiness Directive 96-02-06 (ref. 3), which called for the replacement of the software in that wind-shear detection system on over 1600 aircraft. The directive to change the software applies to a large number of commercial transport aircraft, including the Boeing 727, 737, and 747; the McDonnell Douglas DC-8 and DC-9 series, MD-88, and MD-11 and MD-90-30 series; the Lockheed L-1011-385 series; the Fokker F28 Mark 1000, 2000, 3000, and 4000 series; and the British Aerospace Avro 146-RJ series.

In addition to the safety and reliability issues that are obvious in discussing life-critical systems, the cost involved in software development is an integral issue. According to an article in *BYTE* magazine, "each line of the space shuttle's flight-control software costs NASA contractor Loral about $1000, or ten times more than for typical commercial software" (ref. 4). The much anticipated Advanced Automation System (AAS) that is to replace the FAA's antiquated air-traffic control system is reportedly costing between $700 and $900 per line of code (ref. 5). In addition to the development costs, the costs

of maintenance (e.g., updates, enhancements, corrections, and adaptations to external interfaces) can rival the cost of the initial development. As an example of maintenance cost, the software modification for the wind-shear detection system initially was estimated at about $600 per aircraft, or approximately $1 million.

To confront the growing complexity and quantity of software used in commercial avionics systems (and systems in general), government regulatory agencies such as the FAA and the Department of Defense have mandated the use of certain software development processes and techniques. However, no software engineering method (or combination of methods) has been shown to consistently produce reliable, safe software. In fact, little quantitative evidence exists to show a direct correlation of software development method to product quality. Software verification is the subject of considerable controversy. No general agreement has been reached on the best way to proceed or on the effectiveness of various methods (refs. 6 and 7). Simply put, the knowledge base for engineering software has not reached maturity.

A clear understanding of the entire software development process is essential in defining those methods that may successfully produce quality software. In an effort to increase our understanding of this process, Langley Research Center (LaRC) conducted a series of experiments to generate data to characterize the software failure process (ref. 8). With an increased understanding of the failure process, improved methods for producing reliable software and for assessing reliability can be developed. This paper discusses a project in which the effectiveness of software development methods was examined. The project involved both the development of a framework for conducting scientific experiments and the evaluation of that framework through a case study that involves software guidelines used by the FAA.

This paper is organized as follows. Section 2 provides background information on software engineering experiments, in particular, those conducted at LaRC. Section 3 describes a general framework established to conduct small-scale experiments, and section 4 describes the case study used to test this framework. The analysis of the data from the case study is given in section 5. A project summary and lessons learned are presented in section 6.

2. Background

Computer software allows us to build systems that otherwise would be impossible and provides the potential for great economic gain (ref. 9). The logical constructs of software provide the capability to express extremely complex systems. In fact, computer programs are ranked among the most complex products ever devised by humankind (ref. 6). From the complexity comes the difficulty of enumerating, much less understanding, all possible states of the program, and from that comes unreliability (ref. 10). Identifying unusual or rare conditions is particularly problematic. If a software error exists in a critical system, the cost can be human life (ref. 11).

Although no commercial airline crashes have been directly attributed to software failure, several examples of software errors can be noted that have contributed to the loss of life. One of the earliest examples is the software glitch in the Theratronic International Therac-25, a computer-controlled radiation therapy machine, which caused lethal doses of radiation to be administered to two cancer patients in 1986 (ref. 12). In 1991, during the Gulf War, the Patriot missile defense system failed to intercept an Iraqi Scud missile, resulting in the loss of the lives of 28 American soldiers. Within the Patriot system, a software fault allowed error to accumulate in the calculations that were used to track the missile. Without this fault, the Patriot system might have intercepted the Scud missile (ref. 13). Incidents involving computer systems have become so prevalent that Peter Neumann keeps records of the risks associated with computers and software in a series of monthly articles called "Risks to the Public," which is published on a monthly basis in *Software Engineering Notes* (ref. 14).

Although some members of the software engineering community are quick to announce the latest breakthrough in software engineering technology based on individual success stories, many researchers concur that computer science, especially the software side, needs an epistemological foundation to separate the general from the accidental results (refs. 5 and 15). According to Wiener (ref. 13), "we need to codify standard practices for software engineering—just as soon as we discover what they should be. Regulations uninformed by evidence, however, can make matters worse." Clearly, scientific experimentation is needed to supply the empirical evidence for evaluating software engineering methods.

Although many experiments in software engineering have been undertaken (refs. 16–20), "measurement and experimentation have generally played at best a minor role in computer science and software engineering. It costs a lot of money and effort to do controlled experiments, and that is too high a price for most researchers equipped to do such studies, especially in the world of large-scale software" (ref. 6). Furthermore, most significant software engineering experiments use students in a university setting with relatively small problems and without requiring compliance with any software development standards. The university environment provides a relatively inexpensive and "captive" labor force (in the sense that a student's grade might be directly affected by his participation in the experiment). In the real world, however, labor is expensive, software projects last longer than a 16-week semester, personnel turn over, and real-world problems are quite large and complex.

Software engineering experiments have been conducted over the past 20 years at LaRC with a focus on generating significant quantities of software failure data through controlled experimentation. The Software Error Studies program at LaRC started with a series of studies conducted by the Aerospace Corporation to define software reliability measures and to study existing software error data from a multisensor tracking system and operational avionics systems (refs. 21–23). These initial studies demonstrated clearly that obtaining significant amounts of software error data from fielded systems was difficult at best. These initial studies served as the motivation for conducting experiments to gather software error data representative of a real-world development process.

Following the effort by Aerospace Corporation, Boeing Computer Services (BCS) and the Research Triangle Institute (RTI) conducted several software studies with different applications, including missile tracking, launch interception, spline function interpolation, Earth satellite calculation, and pitch axis control (refs. 24–28). The development process used in these studies generally involved a number of programmers (i.e., n) who independently coded the applications from a given specification of the problem to effectively yield a sample of size n. In these experiments, no specific development standards or life-cycle models were followed. Because the problems were relatively small and simple, the versions were compared to a "gold" (or error free) version of the program to obtain information on software errors.

Although these applications were small and the development process was crude (relative to the process required for obtaining certification on critical software), these early experiments yielded some interesting results (ref. 8). The BCS and RTI studies showed widely varying error rates for faults, that is, evidence that all faults were not equally likely to produce an error. These studies also provided evidence of fault interaction. Fault interaction occurs when "the failure behavior of a program is affected by the presence or absence of two or more faults which either conceal each other . . . or together cause errors when alone they would not . . ." (ref. 8). The failure rates of the different programs also appeared to follow a log-linear trend with respect to the number of faults corrected.

The next project in the Software Error Studies program attempted to build on the data-collection methods used in the earlier experiments and to address some concerns from the previous experiments. To address realism issues about the software application and development process, a guidance and control application based on the control of the Viking lander during its terminal descent trajectory was selected, and the Requirements and Technical Concepts for Aviation RTCA/DO-178B guidelines, *Software Considerations in Airborne Systems and Equipment Certification* (ref. 29), were selected as the

standard. All civil transport airframe and equipment vendors are expected to comply with these guidelines in building systems certified by the FAA for use in commercial aircraft.

As the Guidance and Control Software (GCS) project evolved, two primary objectives were established: to develop a controlled environment for conducting scientific experiments in software engineering and to conduct a case study using the DO-178B guidelines to evaluate the effectiveness of the controlled environment. (The original intent of the GCS project was to conduct an experiment by using multiple implementations of the GCS. However, as the project progressed, it became clear that a true experiment was not possible with the resources available. Consequently, a more reasonable case study became the focus of the project.)

3. Experiment Framework

According to Hamlet (ref. 30), proper operation of a software product is the fundamental measure of quality, that is, "quality software does not fail." Software failure data are clearly important to characterizing software quality. However, little time has typically been devoted to collecting failure data. At best, failure data might include the number of faults identified during the development activities or a classification of faults (according to schemes for categorizing functionality, severity, or other characteristics). Failure data from the actual operation of the software are rarely available. From a statistical perspective, failure data collected during a single operational run represent a single replicate of failure data. By using a variety of statistical reliability models, researchers can make some crude estimates of the reliability of the final software product based on the number of faults that has been removed. Although the determination of the reliability of a software product is interesting research, here we are interested in understanding the effects of software engineering methods on software quality.

A single replicate of software failure data does not provide enough information to make statistical inferences regarding effectiveness of a development method. The original Nagel and Skrivan experiments (ref. 24) established a concept called "repetitive runs" to collect sets of software failure data in a controlled environment so that statistical inferences could be made. In the repetitive run approach, a software implementation (or program) is developed from a specification of the requirements. In a single program run, an implementation is subjected to a series of test cases, and a record is kept of the successive interfailure times following the detection and correction of errors. By using a gold version of the program (i.e., a version that has been shown, usually over time, to be reliable) to generate the expected outcomes for the test cases, a large number of tests can be executed for a single run. Then, a run can be replicated by starting the testing process with the same initial program state and a new series of test cases. This approach can produce sufficient data to make statistical arguments regarding that single product.

This repetitive run concept can be applied to determine the effectiveness of a given development method (or set of methods) with some level of statistical confidence. For a given method, consider a sample (of size n) of software implementations in which each implementation has been developed with that particular method. The repetitive run approach can be used to examine the quality (especially reliability) of each element in the set to determine the effectiveness of that particular method. Unfortunately, finding such a set of software implementations in the real world is essentially impossible. Because software does not fail or wear out as hardware does, a single software implementation is typically developed for a given application. Consequently, evaluating the effectiveness of software engineering methods without actually developing multiple software products is difficult.

3.1. Elements of Framework

Experimentation in software engineering is notoriously difficult in part because the control of variables and environments can be quite daunting (ref. 31). In addition to selecting which software engineering method to study, one must also consider other factors that affect the response of the software to the treatment under study. In defining the development environment, the following factors should be

included: life-cycle model, hardware platform, automated tools, programming language, and experience level of the project developers. The development environment should be controlled to the extent possible to ensure that any difference in measurement is attributable only to the difference in treatment. See Campbell and Stanley (ref. 32) for details on experiment design.

A significant part of the GCS project involved establishing a framework to provide software products to study scientifically. As mentioned above, the framework required an environment for developing and controlling multiple implementations of a given application. The framework also needed to allow for the capture of failure data during development and operation. The framework proposed for the GCS project consisted of three major elements: a software requirements document for a realistic guidance and control application, so that all software implementations were based on the same requirements; a software simulator to run the guidance and control software in a simulated operational environment; and a configuration management system, so that the change process could be controlled and all versions of the implementations could be captured.

Many experiments are comparative; that is, they measure and compare the responses of essentially analogous experimental units after the units have been subjected to some treatment. With this perspective, the requirements for the GCS provided the basis for developing analogous experimental units; the treatment was the application of software engineering methods; and the simulator provided the capability to collect software failure data (responses). The configuration management system captured and controlled the products of the development process.

The key to conducting the experiment was the repetitive run approach. As in all but the simplest problems, a gold version of the GCS was not available. The simulator enabled software failure data to be collected without an oracle or other expensive way to develop expected results. The simulator was designed to run one or more implementations (i.e., up to 28 implementations) of the GCS in a multitasking environment to collect data for comparison. A software failure was indicated when the results from multiple implementations did not agree.

The simulator and data-collection process were strongly dependent on the GCS application. The following sections provide an overview of the GCS application, the GCS simulator, and the configuration management system.

3.1.1. GCS Application

The software application is a guidance and control function necessary to complete the terminal-descent trajectory of a planetary lander vehicle. The original software requirements document for this application, referred to as the GCS specification, was reverse engineered by RTI from a simulation program used to study the probability of success of the NASA Viking Mission to Mars in the early seventies (ref. 33). The software requirements for the GCS focused on two primary needs: to provide guidance and engine control of the lander during its terminal phase of descent onto the planet's surface and to communicate sensory information regarding the vehicle and its descent to an orbiting platform. Figure 1 shows a sketch of the lander during the terminal phase of descent.

The lander includes a guidance package that contains sensors for obtaining information about the vehicle state and environment, a guidance and control computer, and actuators that provide the thrust necessary for maintaining a safe descent. The vehicle has three accelerometers (one for each body axis), one Doppler radar with four beams, one altimeter radar, two temperature sensors, three strapped-down gyroscopes, three opposing pairs of roll engines, three axial engines, one parachute release actuator, and a touchdown sensor. The vehicle has a hexagonal, boxlike shape with three legs and a surface-sensing rod that protrudes from its undersurface.

In general, the requirements for the planetary lander concern only the final descent to the planet's surface. Figure 2 shows the phases of the terminal-descent trajectory of the lander.

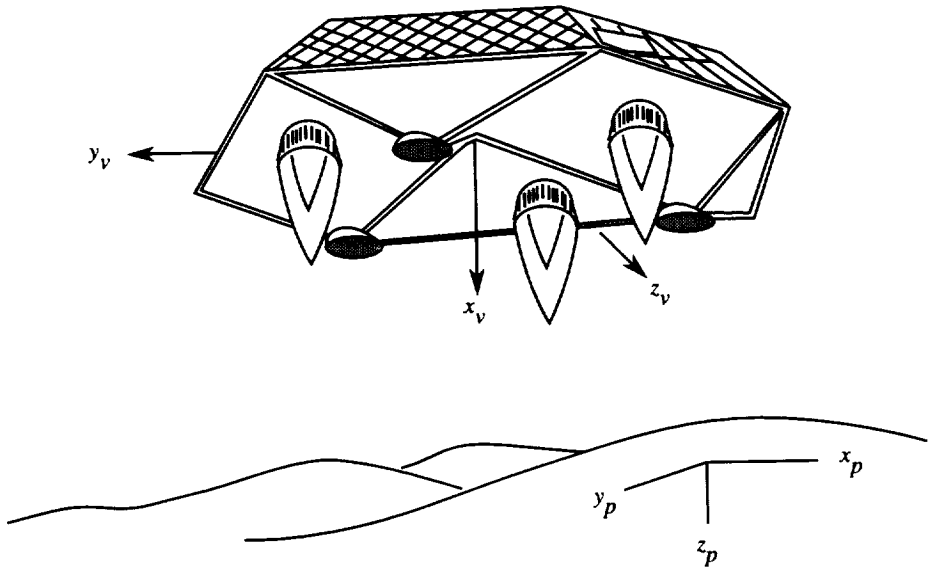


Figure 1. The Viking lander during descent.

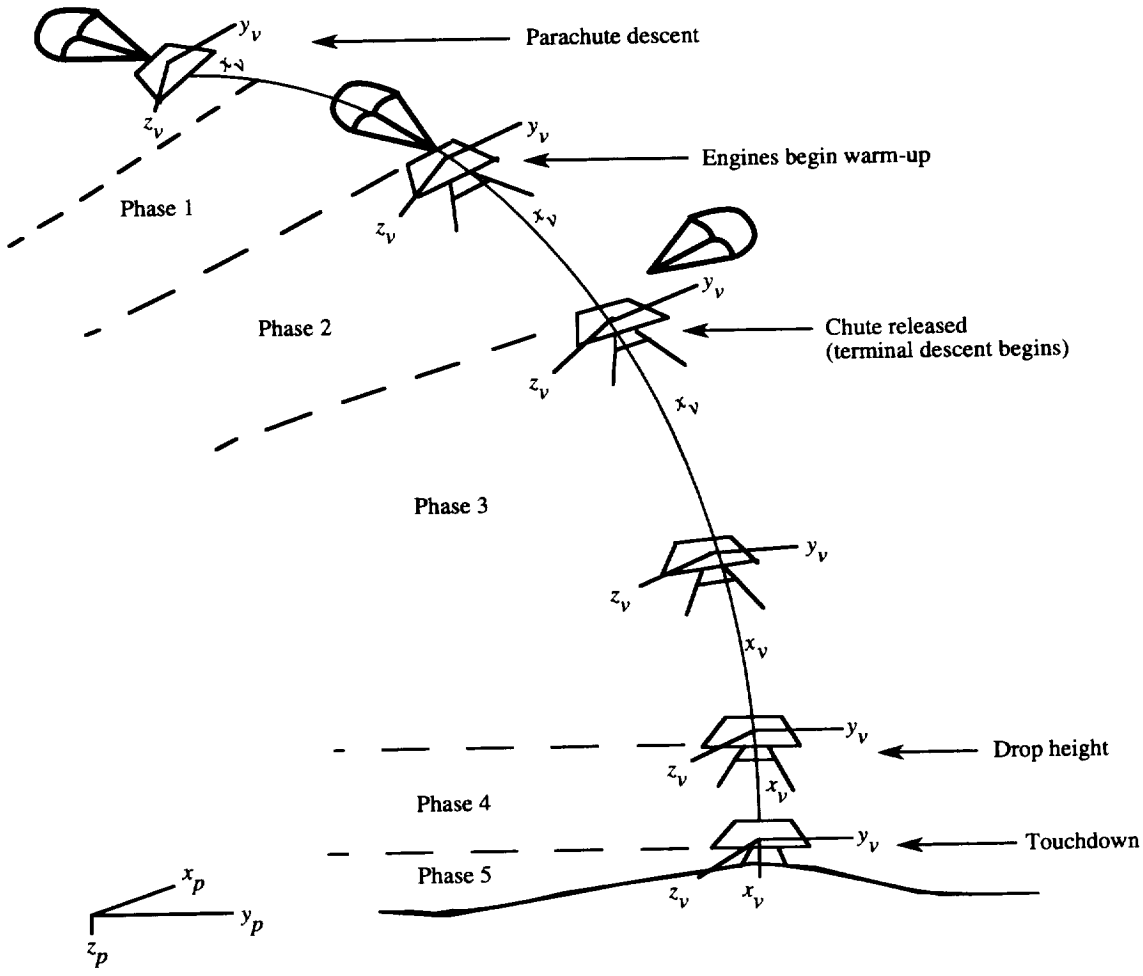


Figure 2. Typical terminal-descent trajectory.

After the lander drops from orbit, the software controls the engines until the vehicle reaches the surface of the planet. The initialization of the GCS activates the vehicle altitude sensor. When a predefined engine-ignition altitude is sensed by the altimeter radar, the GCS begins guidance and control of the lander. The axial and roll engines are ignited; while the axial engines are warming up, the parachute remains connected to the vehicle. During this engine warm-up phase, the aerodynamics of the parachute dictate the trajectory of the vehicle. Vehicle attitude is maintained by firing the engines in a throttled-down condition. After the main engines become hot, the parachute is released and the GCS initiates an attitude correction maneuver and then follows a controlled acceleration descent until a predetermined velocity-altitude contour is crossed. The GCS then attempts to maintain the descent of the lander along this contour. The lander descends along this contour until a predefined engine shut-off altitude is reached or touchdown is sensed. After all engines are shut off, the lander free-falls to the surface.

The RTI engineers used a version of the Structured Analysis for Real-Time System Specification technique by Hatley and Pirbhai (ref. 34) to produce the original GCS specification. In general, this method is based on a hierarchical approach to defining functional modules and associated data and control flows. The computer-aided software engineering (CASE) tool by Cadre Technology, called Teamwork (ref. 35), was also used to refine some of the data and control flow diagrams in the GCS specification.

The structured analysis method was used to decompose the software requirements into 11 major functions called functional units (or process specifications in the terminology of Hatley and Pirbhai (ref. 34)). These functional units were combined to form three subframes, and the three subframes made up a single frame, as show in figure 3.

Approximately 2000 frame iterations were required to complete a single terminal-descent trajectory. Given the control laws specified in the software requirements, the probability that the vehicle would safely land on the planet's surface had to be at least 0.95; that is, given a large number of simulated trajectories, the vehicle should have a 95-percent chance of successfully landing (as opposed to crashing) on the planet's surface. For the original Viking lander, the choice of guidance and control design was based on a statistical approach. Each individual requirement (e.g., successful landing) had to have a probability of at least 0.99. The probabilities were established in terms of technical feasibility, cost, and risk, based on the best available pre-Viking data about Mars (ref. 33). The control laws here were simplified somewhat in comparison with the original Viking control laws for terminal descent; thus, we accepted a slightly smaller probability of success. This criterion for successful landing was

Frame		
Sensor Processing Subframe	Guidance Processing Subframe	Control-Law Processing Subframe
Accelerometer Sensor Processing	Guidance Processing	Axial Engine Control-Law Processing
Altimeter Radar Sensor Processing	Communications Processing	Roll Engine Control-Law Processing
Temperature Sensor Processing		Chute Release Control-Law Processing
Gyroscope Sensor Processing		Communications Processing
Touchdown Landing Radar Sensor Processing		
Touchdown Sensor Processing		
Communications Processing		

Figure 3. Frame, subframe, and functional units.

based on the velocity and attitude at impact and is defined in the GCS simulator. No system-level requirements were available for this application; consequently, no means for tracing the software high-level requirements to the system requirements was available to ensure that all requirements were met. RTI did perform some review of requirements for accuracy, consistency, and verifiability. Version 2.2 of the GCS specification was the first version to be placed under configuration control and constituted the base version of the software requirements for the experiment framework.

3.1.2. GCS Simulator

Each GCS implementation (i.e., code that fulfills the requirements given in the GCS specification) is run in conjunction with a software simulator, which imitates the hardware of the planetary lander to mimic one terminal descent of the lander. The simulator, represented by the large box in figure 4, provides sensor input, based on an expected usage distribution in the operational environment, to the GCS implementation. The GCS implementation initiates the sensor processing, guidance processing, and control-law processing in sequence. At the end of each subframe, data are provided to the simulator to be checked for range violations. At the end of the control-law processing subframe, the simulator performs response modeling for the guidance and control function and formulates new sensor values for the next frame.

A GCS implementation may contain errors that cannot be identified by evaluating only the predefined limits and the final outcome. An obvious example is the computation of an incorrect communication packet. Because the communication packet does not impact the guidance and control function and would not be subject to a limit check, the lander could still land safely; thus, the error would not be identified. The simulator's capability to run multiple implementations in a back-to-back configuration provides a means for identifying these interim failures (or failures that do not affect the final result). In general, given n GCS implementations (all developed to the same requirements document with $n \leq 28$), the back-to-back testing would proceed as follows:

- Start all n implementations with the same input data and allow each implementation to compute the sensor processing in parallel

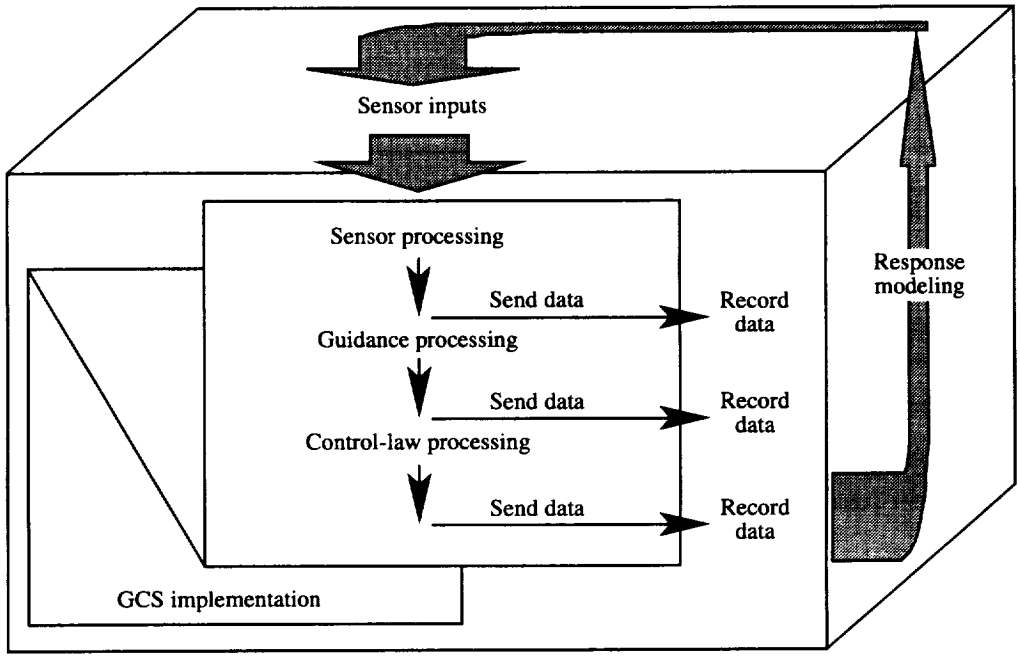


Figure 4. GCS simulator with single GCS implementation.

- Compare the output from each implementation at the end of the sensor processing subframe and record data on any differences or range violations
- Continue through all subframes (checking output at the end of each subframe) until the terminal descent is completed
- Record the landing status for each implementation

An implicit assumption exists with back-to-back testing: when the output is not the same (but is within a given tolerance for "real" data), an error exists. By examining the data from each different result, a set of failure data for each implementation can be identified. These data can be used to characterize the quality of each implementation.

The simulator was implemented on a Digital MicroVAX 3800 computer system with the VAX/VMS V5.5-2HW operating system. To aid the development and testing of the simulator, a prototype GCS implementation called Venus was produced. However, Venus was not developed in compliance with any particular software standard. Although Venus was used extensively to test the simulator and to identify problems in the GCS specification, this prototype was not considered a gold version.

3.1.3. Configuration Management System

A configuration management system is a tool for tracking and controlling the products from all phases of the software development cycle. A good configuration management system is also critical to the experiment framework. The software configuration management process is responsible for providing: a defined and controlled configuration of the software throughout the software life cycle; the ability to consistently replicate a configuration item or to regenerate it if needed; and a known point for review, assessment of status, and control of change by establishing baselines for configuration items.

Within the framework, the configuration environment and activities provide for management of the life-cycle data to develop one GCS implementation and to provide a mechanism for preserving the independence of the life-cycle data for multiple implementations. The Code Management System (CMS) (ref. 36) provides the configuration management for electronic files for the testbed. The CMS is an online library system (located on the MicroVAX 3800 computer system) that tracks the software development process. The CMS library is a VMS directory that contains specially formatted files. The CMS stores files called elements in a library, tracks the changes made to these elements, and monitors access to the elements. An element may contain text, source code, object code, and test cases.

Effective configuration management and data collection mechanisms are essential for obtaining information on the software failure process. For each experiment, a configuration management plan should establish the methods to be used throughout the software life cycle for the software engineering methods under study. At a minimum, the configuration management plan should specify all configuration items and their native format. Each configuration item is placed in a unique CMS library. If an element in a CMS library needs to be modified, then this element must be reserved, changed, and replaced. The original version, or generation, of the element is called generation 1. After an element is reserved and replaced, the element becomes generation 2. All previous generations of any element are easily retrieved from CMS. For more information on CMS, see the CMS User's Guide (ref. 36).

With the CMS, all versions of the software products are preserved. During development activities for a software engineering method, all milestone versions can be saved in the CMS, along with information regarding faults that were identified and removed between versions. Because the various versions of a software product can be regenerated at any time, this configuration management system can serve as a general library of products for other experiments. The next section describes how the GCS development framework can be used in conducting experiments in software engineering.

3.2. Experiment Approach

One form of an experiment is the measurement or observation of responses to some treatment. The process of comparison, or the recording of differences in the responses to various treatments, is fundamental to the collection of scientific evidence. The ability of an experimenter to control the conditions of an experiment can determine the extent to which the differences in responses can be attributed to the treatments. In general, experiments consist of at least four phases: (1) definition, (2) planning, (3) execution, and (4) analysis. Control of the experimental conditions is a factor in phase 3. Although phases 1 and 2 are no different for software studies than for experiments in other fields, the significant costs that can be incurred in phase 3 have contributed to a lack of scientific experiments in software engineering.

The framework established with the GCS components (i.e., the requirements specification, the simulator for operational testing, and the configuration management system) was developed to facilitate experimental studies in software engineering. With the exception of dependence on the GCS application, the framework allows an experimenter complete flexibility in defining phases 1, 2, and 4. An experiment plan may call for the development of GCS implementations in accordance with some software engineering method of interest, or the experimenter may choose to use previously developed products that are stored in the CMS library.

A statistical hypothesis is central to an experiment design. A statistical hypothesis is simply a claim about an unknown attribute of a subject or population of interest. In phase 3, the hypothesis is tested by drawing a sample from the population of interest and making measurements or observations about the elements of the sample. According to Basili, Selby, and Hutchens (ref. 37), this type of software study is especially important "because we greatly need to improve our knowledge of how software is developed, the effect of various technologies, and what areas most need improvement" (ref. 38).

A variety of issues in software engineering are suitable for empirical studies. Pfleeger authored a series of articles that details aspects of experimental design and analysis in software engineering (ref. 38). Some questions that are suitable for analysis include

- If a particular development method is used, will the final product have a certain level of reliability?
- Does test method X detect more software errors than method Y?
- Does the choice of source-code language impact the quality of the code?

Consider the evaluation of the final product for a single software engineering method. One hypothesis of interest could be to determine whether the reliability of the final software product produced with this method is greater than 0.99. To test this hypothesis, multiple GCS implementations could be independently developed in which each implementation is passed through the sequence of activities prescribed by the given method. The final software versions could be tested in the simulator to gather additional failure data and estimates of reliability, which could be used to test the hypothesis. Comparative studies of two or more development methods could be accomplished in a similar manner. To demonstrate that effectiveness of the experiment framework, a pilot case study was conducted. The following section describes this study.

4. DO-178B Case Study

Because one of the goals of the GCS project was to understand how critical avionics software fails, critical software development in the commercial avionics industry was emulated for this case study. The GCS project is a software engineering case study of the DO-178B guidelines (ref. 29). As mentioned earlier, all civil transport airframe and equipment vendors are expected to comply with these guidelines

in building systems for use in commercial aircraft. The following sections provide some background on the DO-178B guidelines and the case study.

4.1. DO-178B Guidelines

Before a new type of aircraft enters commercial operation, the FAA must issue an airworthiness type certificate to the manufacturer. In the United States, the Federal Aviation Regulations (FAR) govern the aircraft certification process. In particular, FAR 25 sets forth the airworthiness standards for civil transport aircraft, including all airborne systems. The FAA issues Advisory Circulars (AC's) in conjunction with the FAR to direct how a manufacturer may demonstrate compliance with the FAR. Although ultimately the manufacturer and the certificating authority must agree on the terms of certification, AC 20-115B states that a manufacturer may follow the DO-178B guidelines as a means for obtaining approval of digital computer software (ref. 39). Consequently, the DO-178B guidelines influence much of the software development for the commercial civil transport industry.

The purpose of the DO-178B document is "to provide guidelines for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements" (ref. 29). The guidance is provided in terms of objectives for each software life-cycle process, where the level of rigor and the amount of life-cycle data are specified according to the level of criticality of the software functions. The DO-178B document defines five levels of criticality (i.e., Level A through Level E). At Level A, anomalous behavior of the software would result in a catastrophic failure condition for the aircraft. At Level E, anomalous behavior of the software would result in no effect on aircraft operational capability.

Because of the importance of the DO-178B guidelines in the development of critical software in the commercial avionics industry, the GCS experiment framework is tested here by developing GCS implementations in compliance with DO-178B. This study provides experience in developing software in accordance with an industry standard. The project was designed as a simple case study rather than an experiment with a statistical hypothesis. The basic concept for the project was to develop the GCS application in compliance with DO-178B and analyze the error data identified during the development process to investigate the quality of the final software products.

The case study involved two independent development teams. (The original project plan called for the independent development of three GCS implementations.) Each team consisted of a programmer and a verification analyst who were employed as professional software developers. The teams were assigned the names Mercury and Pluto; each team was tasked to develop a single GCS implementation from the GCS specification in compliance with the DO-178B guidelines. Each team had auxiliary support from management, software quality assurance, system analysis, and configuration management personnel to meet the objectives delineated in DO-178B.

Table 1 gives a general description of the responsibilities of six major personnel roles defined for the case study. The management, quality assurance, configuration management, and system analysis functions served both development teams. Because the two development teams were to proceed independently through the development processes, special constraints were placed on the level of communication that was allowed among the project participants. In particular, the programmers were not permitted to communicate with each other about their implementations, and the verification analysts were not permitted to discuss specific details about their implementations. Any changes made to the GCS specification during the project, regardless of origin, were broadcast to both development teams.

Table 1. GCS Project Personnel and Organization

Project role	Responsibility
Project leader	Managed all activities of GCS project, including planning, technical direction, and coordination with respect to all life-cycle processes; collected and analyzed data; and scheduled the major milestones to meet goals of project.
Software quality assurance representative	Provided confidence that software life-cycle processes produced software that conformed to requirements by ensuring that project activities were performed in compliance with DO-178B and project standards, as defined in planning documents.
Configuration manager	Provided configuration management of all life-cycle data (documentation, design, code, test cases, and simulator) associated with development of GCS implementations in accordance with DO-178B guidelines and project standards.
System analyst	Provided expertise regarding software requirements for GCS (described in GCS specification) to project participants and maintained the GCS specification in accordance with DO-178B guidelines and project standards.
Programmers (one for each team)	Independently developed one implementation of GCS in accordance with GCS specification, DO-178B guidelines, and Software Development Standards. Development included generation of detailed design description, source code, and executable object code.
Verification analysts (one for each team)	Defined and conducted all verification activities associated with development of one GCS implementation in accordance with GCS specification, DO-178B guidelines, and Software Development Standards.

4.2. Life-Cycle Processes

This section provides an overview of the life-cycle activities for each of the GCS implementations. The DO-178B guidelines define three types of software life-cycle processes: the software planning process, the software development processes, and the integral processes. In the software planning process, the software development processes and the integral processes are defined and coordinated. The software development processes involve identification of software requirements, the software design and coding, and integration; that is, the development processes directly result in the software product. Finally, the integral processes function throughout the software development processes to ensure integrity of the software products. The integral processes include the software verification, configuration management, and quality assurance processes.

During the course of a software life cycle, data are produced to manage and plan the life-cycle activities and to document the results of those activities. Life-cycle data, as defined in section 11.0 of DO-178B, are necessary to provide the FAA with evidence that the life-cycle activities comply with the guidelines. For the purposes of this case study, each development team was required to follow the same life-cycle development plan. The documents produced during the life-cycle activities for the Mercury and Pluto implementations of GCS are listed in table 2.

The following sections give a general overview of the life-cycle activities for the development of the GCS implementations.

4.2.1. Software Planning Process

The objective of the software planning process is to define the development and integral processes necessary to produce a software product that satisfies the given requirements (i.e., the GCS specification). Thus, the primary activity during this process was to document the plans for all activities in the life-cycle processes, including the flow and transition criteria; the development environment, including methods and tools to be used; and the development standards. Table 3 shows the objectives for the planning process based on the tables in annex A of DO-178B.

Table 2. Life-Cycle Data for GCS Project

Software life-cycle process	Life-cycle data general to both GCS implementations	Life-cycle data unique to each implementation
Software planning and management	• PSAC • Software development standards • Software configuration management plan • Software quality assurance plan • Software verification plan • Software accomplishment summary	
Development: Software requirements	• GCS specification (part of experiment framework)	
Software design		• Design description
Software coding		• Source code
Integration		• Executable object code
Integral: Verification	• Software verification cases and procedures (includes requirements-based test cases)	• Software verification results (includes structure-based test cases)
Configuration management	• Software configuration index • Software configuration management records • Support documentation change reports	• PR's
Quality assurance	• Software quality assurance records	

Table 3. Activities and Products of Planning Processes

Objectives	Major activities	Products
Define development and integral processes: Transition criteria Life cycle Project standards	• Develop project planning documents to comply with DO-178B	• PSAC • Software development standards • Software verification plan • Software configuration management plan • Software quality assurance plan

The *Plan for Software Aspects of Certification* (PSAC) is one of the most important data items because it defines the methods that have been established to produce the development products (e.g., design, source code, and executable object code) in compliance with DO-178B. The certificating agency uses the PSAC to determine whether the proposed project plan is sufficiently rigorous for the level of software being developed. For this project, the GCS was considered to be Level A software.

The standards for the development products and other project documentation are given in the *Software Development Standards*. This document also contains a description of tools and methods to be used during the development phase (e.g., programming language). Other fundamental information about project procedures (e.g., configuration management and problem reporting) is also addressed in the *Software Development Standards*; thus, this document served as a single handbook for project participants.

Because both teams were required to follow the same development and integral processes, only one set of planning documents (i.e., the PSAC, which includes the software development plan, the *Software Verification Plan*, the *Software Configuration Management Plan*, and the *Software Quality Assurance Plan*) was developed, along with a single *Software Configuration Index*. Most of the remaining life-cycle data were specific to each implementation.

Figure 5 gives an overview of the life-cycle activities that were defined in the PSAC. The software development processes followed a modified waterfall life-cycle model that started with a limited requirements process (limited in the sense that the requirements were given as part of the experiment framework) and continued through the design, code, and integration processes.

Verification activities correspond with each of the four development processes. All artifacts (i.e., products) produced throughout the life-cycle activities were controlled through the configuration management process. Some significant artifacts are shown in figure 5. The quality assurance process provides procedures for monitoring the life-cycle processes and reviewing the project artifacts to ensure conformance with the plans and standards defined in the planning process. Because of resource limitations, a certification liaison process, as defined in DO-178B, was not possible for this study.

4.2.2. Software Development Processes

Two development teams worked independently to produce two implementations of the GCS. Each programmer was responsible for deriving a software design that consisted of low-level requirements and software architecture from the GCS specification and for translating that design into source code. Each verification analyst was responsible for conducting the verification activity for each artifact until the transition criteria, defined during the planning process, were satisfied. Figure 6 shows the verification activities and transition criteria associated with the artifacts for each development process.

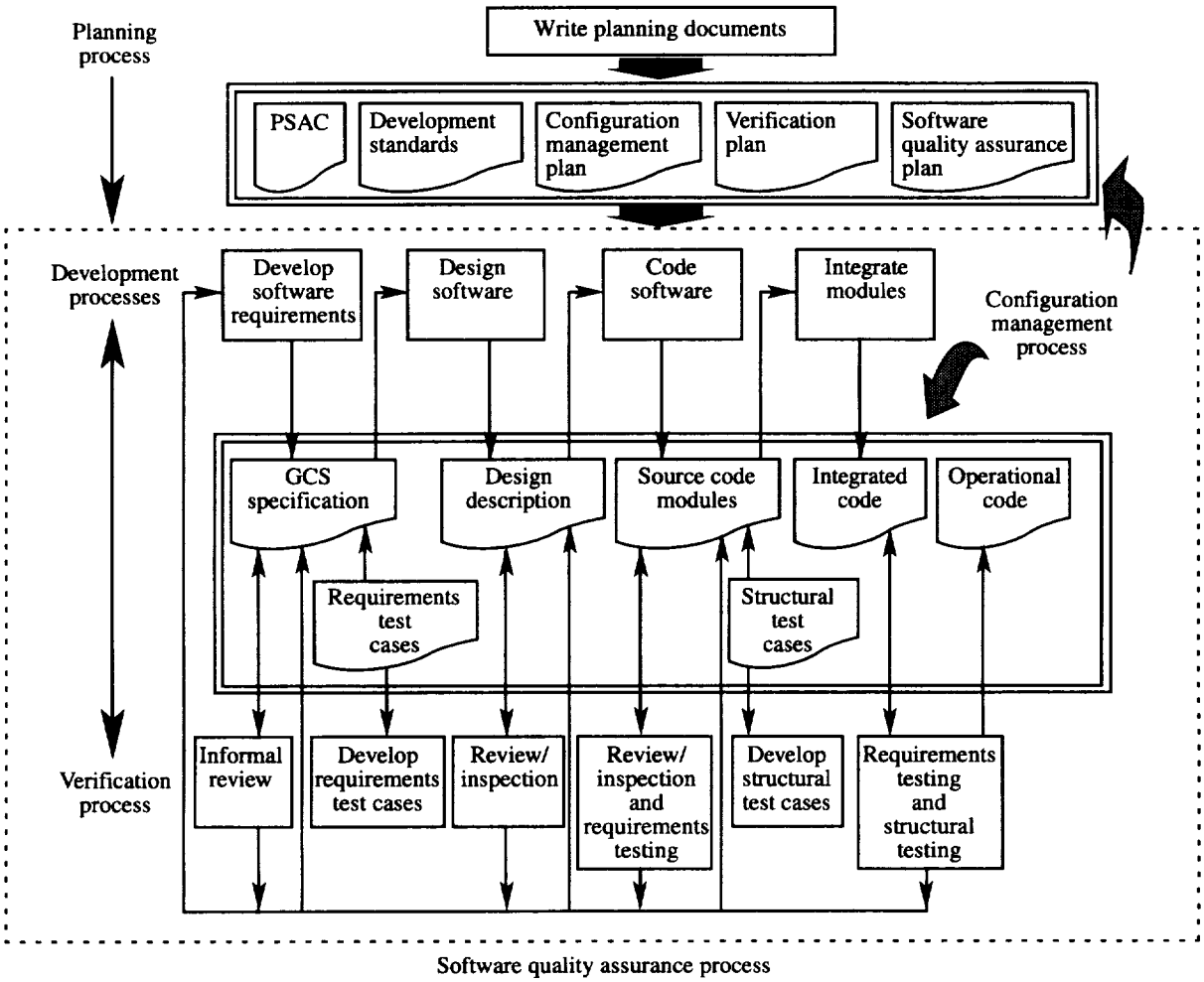


Figure 5. Life-cycle activities flow for GCS project.

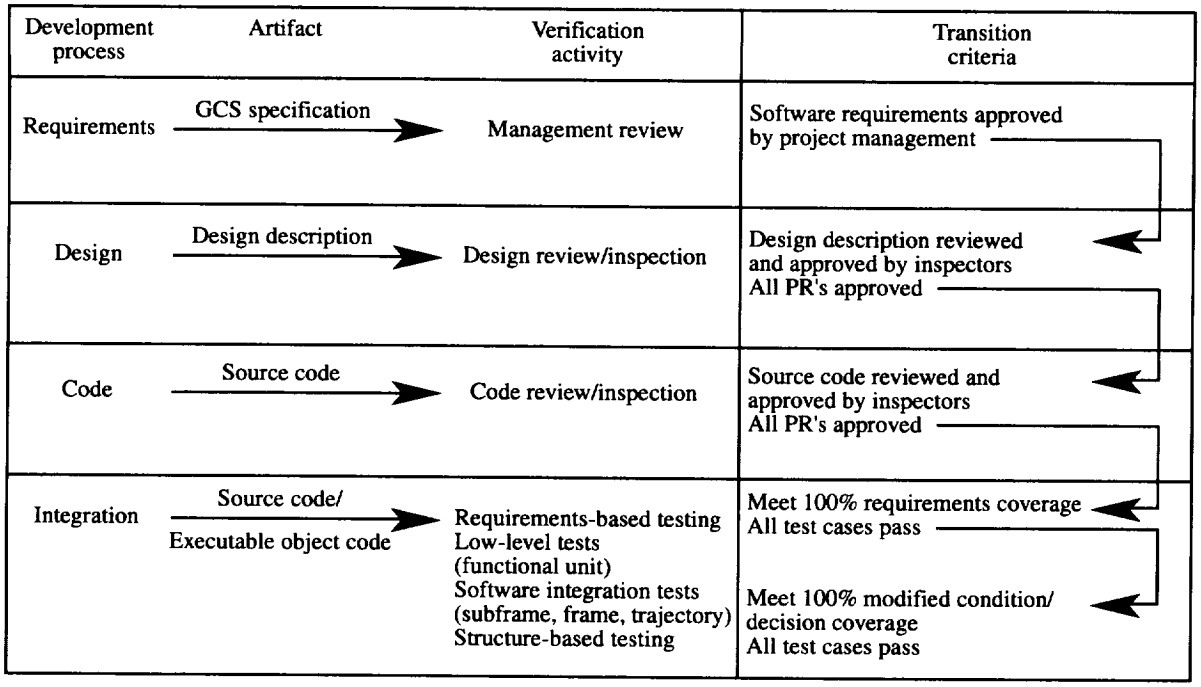


Figure 6. Overview of verification activities and transition criteria for development processes.

Generally, the criterion for transitioning to the next development process was the successful completion of the verification activity (fig. 6). Because the purpose of the verification activities was to detect and report errors introduced during the development, a change-reporting system was established to report problems and track changes made to the life-cycle data. An effective system for the reporting and tracking of problems was also extremely important in terms of project goals because one of the primary objectives of the case study was to collect software failure data. In the context of this case study, a problem was defined as any question or issue that was raised for consideration, discussion, or solution regarding some artifact of the software development process.

A system of problem reports (PR's) and action reports (AR's) (see the appendix) was created to document the problems and any subsequent changes that were made to the development products. A PR contained the following information: the stage in the development process in which the problem was identified, the configuration identification of the artifact, a description of the problem (e.g., noncompliance with project standards or output deficiency), and a history log for tracking the progress and resolution of the problem. All problems were investigated to determine if an actual fault was detected, in which case corrective action was taken and documented on the AR form. Each AR contained the configuration identification of the affected artifact (based on the labels used in the CMS libraries) and a description of the change that was made. Change control procedures, as described in the *Software Configuration Management Plan*, were followed when an actual change was made to a configuration item. When no change was required in response to the PR, the AR form contained the justification for not making any changes. A PR was considered to be closed or completed when the person who identified the problem and the software quality assurance representative both signed the PR form to indicate that the change had been reviewed and was deemed correct.

The change-reporting system was developed to comply with the DO-178B guidelines for problem reporting. As a result, the change-reporting system for this case study was not designed from an analysis perspective; thus, the data were not categorized by type or severity or other characteristic. In fact, a conscious effort was made not to categorize data on the report forms for a number of reasons: it detracted from the purpose of the form (which was to call attention to a problem and get it fixed); we did not want

to spend development time in analysis activities; different people categorize errors differently (i.e., much of data analysis can be subjective); and a single problem and action may not have captured the complete problem or all actions necessary to fix the problem. In fact, a number of problems (especially those regarding changes in the GCS specification) were not completely resolved on a single PR. Consequently, analysis of the PR data was reserved until after the development activities, so that the data set could be analyzed as a whole.

The following sections give an overview of the development and corresponding verification activities for the requirements, design, code, and integration processes. A discussion of the problems identified in each phase is given in section 5.

4.2.2.1. Software requirements process. The primary objective for the software requirements process is to develop high-level requirements that are accurate, consistent, and verifiable. These requirements must be both traceable to and in compliance with the system requirements. In this case study, however, the software requirements, in the form of GCS specification version 2.2, were given as part of the experiment framework. From the GCS specification, a total of 128 requirements were specified in a requirements traceability matrix. This requirements traceability matrix was used to trace each software requirement in the design and in the code and to identify test cases that corresponded to a given requirement, as required by DO-178B. Table 4 shows part of that matrix for the Temperature Sensor Processing (TSP) functional unit.

Because this case study represented the first implementation of the software requirements, we assumed that changes would be made to the requirements as the Pluto and Mercury implementations developed. To ensure that all questions and problems regarding the specification were addressed, a formal communication protocol was established. All questions about the GCS specification were addressed to the system analyst through an electronic conferencing system called VAX Notes. The system analyst then determined whether the GCS specification should be modified and initiated a change report if necessary. A summary of those changes is given in section 5.1.

The software design process started with the release of version 2.2 of the GCS specification. The following section describes the process, as well as the results for the Mercury and Pluto implementations.

4.2.2.2. Software design process. The major DO-178B objective of the software design process is to refine the software high-level requirements into software architecture and low-level requirements that can be used to implement the source code. A detailed design description should be a complete statement of the software low-level requirements and should address exactly what must be accomplished to meet the objectives stated in the GCS specification; that is, the detailed design should contain an algorithmic solution. The low-level requirements should be accurate, consistent, verifiable, traceable to the high-level requirements, and directly translatable into source code with no further decomposition required.

Table 4. Requirement Traceability Matrix for TSP

Functional requirements	Design	Code	Test cases
2.1.5 Temperature Sensor Processing			
2.1.5-1 Calculate solid-state temperature			
2.1.5-2 Calculate thermal temperature			
2.1.5-3 Determine which temperature to use (solid state or thermocouple)			
2.1.5-3.1 Calculate the thermosensor upper limit			
2.1.5-3.2 Calculate the thermosensor lower limit			
2.1.5-4 Determine atmospheric temperature			
2.1.5-5 Set status to healthy			

Both programmers used the structured analysis and design methods described by Hatley and Pirbhai (ref. 34) to generate a design description. Further, each programmer was required to use the Teamwork tool to develop a structured design, and the output from Teamwork was required for the verification activities. The *Software Development Standards* provided guidance on the design procedures and on the use of Teamwork for the design descriptions.

The objectives of the software verification were to verify that the low-level requirements and software architecture complied with the high-level requirements and were accurate, complete, consistent, and verifiable. A series of inspections based on the work of Fagan (ref. 40) was used to review the design descriptions. A unique review team was used for each implementation. Each review team consisted of the programmer and the verification analyst assigned to the implementation, the system analyst, and the software quality assurance representative. All members of the review team, except for the software quality assurance representative, participated as an inspector. Each inspector performed a critical reading of the product to identify any defects. The inspectors used the review procedures (defined in the *Software Verification Cases and Procedures* document), the design review checklist, and the inspection logs to aid in the review. After the inspectors completed their individual reviews of the product, group inspection sessions were held to discuss the product and defects. The requirements traceability matrix was also completed to trace the low-level requirements in the design description to the high-level requirements in the GCS specification. The *Software Verification Cases and Procedures* document gives further details on the design review process.

Preliminary and final design reviews were held for each implementation. Several inspection sessions (limited to 2 hr each) were needed for each review. After each review, PR's were written for all items that were identified as problems by the inspection team. The original intent of the problem-reporting process was to capture each individual problem on a separate PR. However, early in the inspection sessions, a large number of problems were identified. Because the problem-reporting system was a paper-based system, project management made the regrettable decision to allow multiple problems to be recorded on a single PR to save time and effort. Thus, a one-to-one correspondence between the number of PR's and the number of errors does not exist. However, because the resolution of all PR's was the transition criterion for this phase, the number of PR's was noted. During this phase, 19 PR's were issued to the Pluto programmer, and 22 PR's were issued to the Mercury programmer. After all PR's were resolved (i.e., the problem was corrected and the correction was approved by the software qualitive assurance representative), the programmer was allowed to develop the source code.

4.2.2.3 Software code process. The major DO-178B objective of the software coding process is to produce a source code that is traceable, verifiable, consistent, and correctly implements the low-level requirements given in the detailed design description. The GCS specification was written with the assumption that a GCS implementation would be coded in the FORTRAN language; however, the framework does not preclude the development of GCS implementations in other programming languages. For this case study, the GCS implementations were coded in VAX/VMS FORTRAN because the host system for the software was a VAX/VMS system; VAX FORTRAN, an implementation of the full FORTRAN-77 language, conforms to the American National Standard FORTRAN, ANSI X3.9-1978. Programmers were instructed to use structured programming techniques whenever practical and were instructed not to use unconditional GO TO statements. No further limits were placed on the use of the features of the VAX FORTRAN language, including the VAX FORTRAN extensions.

Some metrics on the size of the source code and the executable object code for each implementation are given in table 5. Because each implementation was designed to run only in conjunction with a software simulator that was instrumented to collect data to support the research (as opposed to having resource restrictions imposed by a larger system), no special timing or memory requirements were specified for the software.

Table 5. Software Characteristics

Software characteristic	Mercury	Pluto
Noncommented lines of source code	1747	2232
Executable object code size	32 768 bytes	24 768 bytes

In a conventional coding process, a programmer is allowed to compile and execute his code as progress is made in the development. However, for this case study, the programmers were not allowed to execute the code. After the programmer completed and clearly compiled the source code, the code was released to the verification process. A Fagan type of inspection (ref. 40) was conducted on the source code for each implementation; the procedures used were the same as those used for the design inspections. Three PR's were issued to the Pluto programmer and two were issued to the Mercury programmer as a result of the inspection process. After all PR's were resolved, the verification analyst started the integration process.

4.2.3. Integration Process

The integration process for the GCS implementations consisted of software integration. (Because no actual hardware was required for this project, no hardware/software integration was necessary.) The corresponding verification activity was to test the code to ensure compliance with high- and low-level requirements and to ensure robustness. The DO-178B guidelines recommend (but do not require) that a multilevel testing plan be implemented in order to meet all the requirements and the structural coverage objectives. A bottom-up approach was used to test the implementations; thus, testing began at the functional unit level and moved up through the subframe, frame, and trajectory levels. For Level A software, the testing must achieve 100 percent requirements coverage and 100 percent modified condition/ decision coverage (MC/DC).

The overall objective of requirements-based testing was to show that the software provided the specified functionality without adding extra functionality. In the interest of conserving time and other resources, the verification analysts in the case study worked together to produce a single suite of requirements-based test cases from the GCS specification before the source code was produced (so that no bias was introduced from the different implementations). Test cases were generated using equivalence-class partitioning and boundary-value analysis techniques, as described by Myers (ref. 41). Equivalence classes were determined for each variable defined in the data dictionary in the GCS specification. Each normal-range test case was created by choosing inputs from only the valid equivalence classes, and each robustness case was created by choosing a single input from an invalid equivalence class and selecting all other inputs from valid classes. Mathematica, of Wolfram Research, Inc. (ref. 42), was used to compute the expected results for all test cases. The test cases were traced to the low- and high-level requirements through the requirements traceability matrix. Only one PR was issued for the Mercury implementation as a result of requirements-based testing, and four PR's were issued for the Pluto implementation.

Once the requirements-based testing and all PR's were completed, each implementation was analyzed for structural coverage. Each verification analyst was responsible for the structural coverage analysis of his own implementation. During the analysis, a directed graph of the structure of each source-code module was generated using McCabe's Analysis of Complexity Tool (ACT) (ref. 43). Figure 7 shows the directed graph produced by the ACT for the TSP functional unit.

For each functional unit, the verification analyst examined the paths and decision nodes in the structured graph and the cases in the requirements-based test suite to identify any remaining structures that

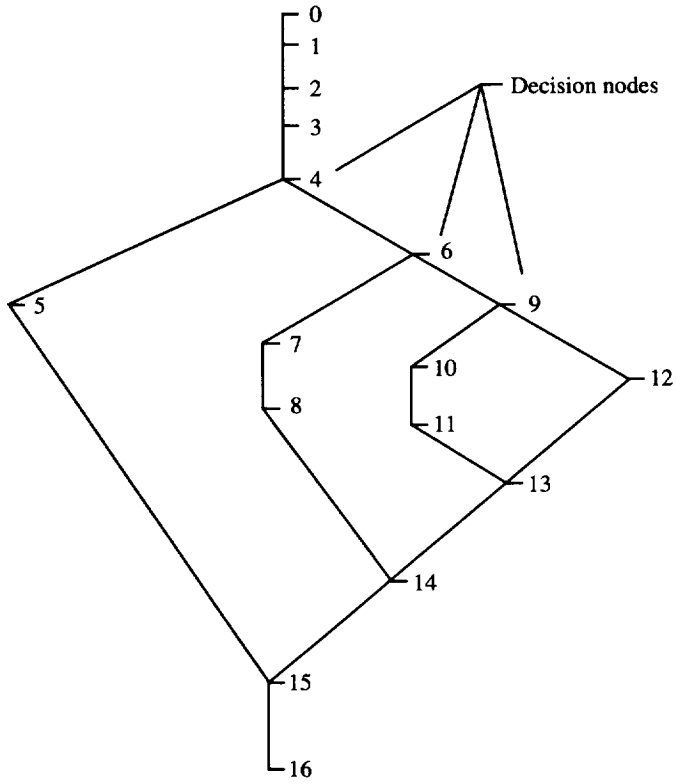


Figure 7. Structure graph for TSP functional unit.

needed to be exercised to meet the structural coverage requirement. To achieve 100 percent MC/DC, a sufficient number of test cases needed to be examined to ensure that

1. Each decision takes on every possible outcome at least once
2. Each condition in a decision takes on every possible outcome at least once
3. Each condition is shown to independently affect the decision outcome
4. Each entry and exit point is invoked at least once

The method of using decision and pairs tables, as discussed by Chilenski and Miller (ref. 44), was used to represent the states that should be covered to satisfy the MC/DC requirement. The decision table for the TSP functional unit, given in table 6, shows the conditional statements for each decision node in the directed graph in figure 7; table 6 also lists the test cases to show that each decision takes on every possible outcome at least once.

Table 6. TSP Decision Table

Graph node number	TSP decisions	True output test cases (a)	False output test cases (a)
4	(SOLID_STATE_TEMP.LT. LOWER_PARABOLIC_TEMP_LIMIT).OR. (SOLID_STATE_TEMP.GT. UPPER_PARABOLIC_TEMP_LIMIT)	TSP_NR_002	TSP_NR_001
6	THERMO_TEMP.LT. M3	TSP_NR_006	TSP_NR_001
9	THERMO_TEMP.GT. M4	TSP_NR_007	TSP_NR_001

[a]The labeling system for the test cases includes the module name, indication of the type of test case (NR for normal range and RO for robustness), and test case number. For example, TSP_NR_001 is the first normal range test case for the TSP functional unit.

Because multiple conditions exist for the decision at node 4, a pairs table is used to show that each condition in the decision takes on every possible outcome at least once and that each condition independently affects the decision outcome. Table 7 shows the pairs table that corresponds to decision node 4 in the directed graph for TSP.

To show whether a condition independently affects the final decision, the outcome of that condition can be varied while all other conditions remain fixed. In table 7, the asterisks in the columns labeled "Independent of condition" indicate those test cases that demonstrate independence for each condition. For example, the combination of test cases `TSP_NR_001` and `TSP_NR_002` shows that condition 1 independently affects the final decision.

Finally, table 8 gives the test cases that address the entry and exit conditions for the modules within the TSP functional unit.

In the case of the TSP functional unit, test cases from the requirements-based test suite covered all necessary conditions for satisfying the MC/DC criteria. For those functional units for which the code structure was not adequately covered, new test cases were created. Table 9 shows the number of test cases required to meet the objectives for the integration process.

Table 7. MC/DC Pairs Table for Decision Node 4 of TSP

`SOLID_STATE_TEMP.LT. LOWER_PARABOLIC_TEMP_LIMIT` (Condition 1) (a)	`SOLID_STATE_TEMP.GT. UPPER_PARABOLIC_TEMP_LIMIT` (Condition 2) (a)	Final decision	Test case	Independent of condition (b)	
				1	2
0	0	0	`TSP_NR_001`	*	*
0	1	1	`TSP_NR_003`		*
1	0	1	`TSP_NR_002`	*	

[a] 0 = False value for the condition; 1 = True value for the condition.
[b] * = Test cases that demonstrate independence for each condition.

Table 8. MC/DC Entry and Exit Test Cases for Modules in TSP

Module	Test case
`LOWER_PARABOLIC_FUNCTION`	`TSP_NR_001`
`UPPER_PARABOLIC_FUNCTION`	`TSP_NR_001`

Table 9. Number of Test Cases for Integration

Type of test case	Number of test cases
Requirements-based test cases	445
Functional unit	379
Subframe	23
Frame	9
Trajectory	34
Structure-based test cases	
Mercury	33
Pluto	38

The requirement for demonstrating 100 percent MC/DC for Level A software is a major issue for avionics software vendors. A number of vendors claim that the additional analysis and testing that is required to meet 100 percent MC/DC is very expensive and does not reveal additional errors. Others claim that conducting sufficient tests to meet the MC/DC requirement provides reasonable assurance that the code structure has been completely exercised and the errors have been removed. In this case study, approximately 7–8 percent of the total test suite resulted from the structural coverage analysis. The structure-based testing did not identify any problems in either implementation. Further analysis of the Pluto team's structure-based test cases showed that 21 of the 38 test cases should have been covered in requirements-based testing and that the remaining 17 could have been detected in requirements-based testing if the low-level requirements defined in the design had been added to the traceability matrix.

Ideally, if requirements-based testing is done with very low-level requirements (including the low-level requirements identified in the design), few (if any) additional test cases should be needed to achieve 100 percent MC/DC. Thus, the importance of clearly enumerating each low-level requirement throughout the development process and of tracing each requirement to all appropriate test cases is emphasized. Establishing this trace by hand is a tedious and error-prone process. Restricting the number of conditions allowed in a decision in the project design and coding standards can improve the verification effort (i.e., inspections and testing) and the coverage analysis. Automated tools could also be helpful.

Because the MC/DC requirement concentrates on identifying faults that lie in Boolean expressions, a more careful approach to requirements-based testing may eliminate the need for demonstrating 100 percent MC/DC; further analysis and experimentation are needed.

5. Analysis of Data From Case Study

Much of our understanding regarding the effectiveness of a development method comes from examining the changes that were made to the project artifacts throughout the life cycle, in particular, those changes made to the requirements and to the final source code. A development method is thought to be more effective if the preponderance of errors is identified at the earliest phases of the life cycle. That is, for a development method that spans the requirements, design, and coding processes, the more errors that are identified and eliminated at the requirements phase, the better.

Papers on software engineering and process improvement studies often are filled with graphs and tables that indicate the number of errors found and that neatly classify those errors. Unfortunately, counting errors is difficult. For this study, we use the following definitions:

Problem: any question or issue raised for consideration, discussion, or solution regarding an artifact of the software development process. Problems are brought to the attention of project members through the change reporting system; that is, whoever identifies a problem should initiate a PR. Most PR's are issued as a result of a verification activity.

Error: with respect to software, a mistake in requirements, design, or code. In accordance with DO-178B, any aspect of the requirements, design, or code is judged to contain an error if it is

- Ambiguous (information is ambiguous if more than one interpretation is possible).
- Incomplete (information is incomplete if it does not include all necessary, relevant requirements and/or descriptive material; if responses are not defined for the range of valid input data; if figures are not properly labeled; or if terms and units of measure are not defined).
- Incorrect (information is incorrect if it does not correctly implement the required function).
- Inconsistent (information is inconsistent if conflicts exist within).
- Not verifiable (information is not verifiable if it cannot be checked for correctness by a person or tool).

- Not traceable (information is not traceable if the origin of its components cannot be determined).
- Not modifiable (information is not modifiable if it is structured or has a style such that changes cannot be made completely, consistently, and correctly while retaining the structure).

Fault: a manifestation of an error in software. A fault may cause a failure.

Failure: the inability of a system or system component to perform a required function within specified limits. A failure can result when a fault is encountered.

When a problem is initially detected and an error is identified, the scope of that error is not always clear. A series of PR's might be necessary before the entire scope of an error becomes obvious. Consequently, for errors that are not simple, a single change report often does not completely reflect the entire correction. Also, an error may be clearly identified in a PR; however, the programmer may only implement a partial fix, and the quality assurance process may fail to detect that the fix is not complete. Consequently, a one-to-one correspondence between the number of change reports (or the problems delineated in a change report) and the number of errors is virtually impossible.

Examining the errors that were made in each of the project artifacts and how they were identified and corrected can only be done with accuracy after the development process is complete. Intermediate versions of the design and code must often be examined to understand each change, and the entire set of change reports may need to be reviewed to gather all related changes to clearly identify each error. Even with all the change reports and intermediate versions of the artifacts, clear identification of the errors may not be easy. Change reports for a single error can trace through the requirements, design, and code.

Throughout the life-cycle processes, changes made to the requirements, design, and code were recorded using the problem-reporting systems established for the project. This section contains brief summaries of the problem reports issued for the GCS specification and the Mercury and Pluto implementations of the GCS.

5.1. Summary of Requirements Changes

During the case study, many changes were made to the GCS specification to correct errors and improve organization, grammar, and punctuation. A total of 29 change reports were issued against version 2.2 of the GCS specification and 7 were issued against version 2.3. Remember that each change report can actually represent multiple problems, errors, and changes. After the release of version 2.2 to the programmers for the design process, a significant number of changes were made to the GCS specification as the system analyst became more experienced with Hatley and Pirbhai's method of structural analysis and the Teamwork tool (ref. 34). Following release of version 2.3 of the GCS specification, the specification was considered stable. As shown in figure 8, most changes to the specification were made while the programmers were developing their designs.

Figure 8 shows that most of the changes to the GCS specification occurred early in the development; however, the figure does not provide information regarding the types of errors that resulted in changes. The GCS specification contains two types of information: information about the functionality that a GCS implementation must provide and information about the context or environment of those requirements. A single change report could identify a number of problems in either the functional requirements or the context of those requirements. A context change was defined as any change that did not impact the functionality. For example, throughout this project, a great deal of ambiguity was evident in the high-level structured analysis charts. As understanding of Hatley and Pirbhai's method (ref. 34) and the Teamwork tool increased, the charts were refined in a series of change reports to eliminate ambiguity. Because the high-level charts only provide an overview of the GCS system without actually defining functionality, those errors that were attributable to ambiguity in the high-level charts were considered to be context errors.

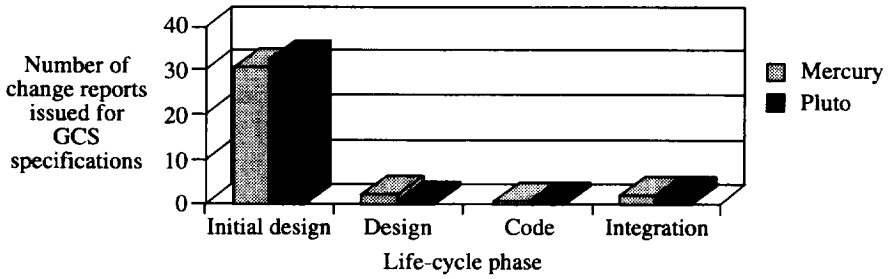


Figure 8. Change reports issued for GCS specification per life-cycle phase.

The errors identified in the functional requirements are the most interesting. Table 10 addresses the reason for each change that was made in functional requirements, the impact of that error with respect to the expected output, and the impact of each change on the two GCS implementations.

Most of the changes made to the GCS specification were made to clarify requirements that would not be likely to impact the successful landing of the vehicle. For example, version 2.2 of the specification required that the two functional units that dealt with radar sensors (i.e., Altimeter Radar Sensor Processing (ARSP) and Touchdown Landing Radar Sensor Processing (TDLRSP)) be executed only during even-numbered frames. Because no clear reason was evident for this requirement (i.e., the requirement was not traceable to any other requirement), the scheduling requirement was modified (via Support Documentation Change Report (SDCR) 2.3-2.1) so that all functional units in the sensor processing subframe executed during every frame. The change to correct this error in the requirements initiated changes in the parts of the GCS specification that related to that function, namely, the scheduling section and the process specifications for the ARSP, the TDLRSP, and the Communications Processing (CP). This change also initiated changes to the Mercury and Pluto designs.

After the design phase, only two requirements errors were identified that could have impacted mission success. Both errors involved equations in which valid input would have caused a negative square root calculation; these errors were not identified until the last phase of integration testing for the Pluto implementation. The first error involved a requirement to calculate the standard deviation of three consecutive accelerometer readings. A specific equation was given in the GCS specification for this calculation. (The argument can be made that giving an equation for the standard deviation in a requirements specification is not appropriate. Although our programmers were professionals, we did not expect them to have a knowledge of statistics and, thus, felt that the inclusion of an equation was reasonable for this case study.) When no change occurred in the reading, the standard deviation was zero. However, because of precision limitations, the form of the standard-deviation equation that was originally specified could yield a negative square root. The requirement was changed to state that no calculation of the standard deviation was necessary when the accelerometer reading did not change; thus, the possibility

Table 10. Summary of Errors Identified in GCS Specification

Error characteristic	Number of errors	Number of errors that impacted outputs	Number of errors that impacted system failure
Ambiguous	11	11	2
Incomplete	3	1	1
Incorrect	5	5	2
Inconsistent	3	2	0
Not traceable	1	0	0
Not verifiable	1	1	0
Other	1	1	1
Total	25	21	6

of a negative square root was eliminated. The design and code for both GCS implementations were changed in response to this requirements change.

The second significant error was the most interesting. During guidance processing, some calculations are made to determine whether to turn the axial and roll engines off and to determine the guidance processing phase. Both situations require the determination of whether the vertical velocity of the vehicle is above or below some maximum vertical velocity for safe landing (MAX_NORMAL_VELOCITY). The variable GP_ALTITUDE is the altitude with respect to the planet's surface as seen by the guidance processor. The following condition was specified originally:

$$\sqrt{2*\text{GRAVITY} * \text{GP\_ALTITUDE}} + x \text{ component of GP\_VELOCITY} \leq \text{MAX\_NORMAL\_VELOCITY}$$

Testing of the Pluto implementation revealed that under certain conditions the lander could hit the planet's surface with sufficient momentum to cause GP_ALTITUDE to be negative; that is, the lander would dig into the planet's surface. The two test cases that revealed this error stressed the maximum vehicle velocity in the y direction and the maximum vehicle velocity in the z direction. If GP_ALTITUDE were negative, then the square root in the velocity calculation would be undefined and, consequently, system failure would occur. To eliminate the possibility of a negative square root, the conditional equation was modified as follows:

$$\sqrt{2*\text{GRAVITY} * \text{maximum(GP\_ALTITUDE,0)}} + x \text{ component of GP\_VELOCITY} \leq \text{MAX\_NORMAL\_VELOCITY}$$

Change report 2.3-7 was issued to implement the change in the GCS specification. Because the software requirements changed, the design and source code for both the Mercury and Pluto implementations were changed. The PR 27 for the Mercury implementation and the PR 30 for the Pluto implementation describe the changes that were made to those implementations.

The next section discusses the changes that were made to the Mercury and Pluto implementations of the GCS.

5.2. Summary of Changes to GCS Implementations

Each GCS implementation independently went through three distinct phases (i.e., designing, coding, and testing) in the development of the executable code. The change process for a GCS implementation in this case study officially started when the software design description was submitted for initial inspection. After a design was submitted for inspection, the design description was placed under configuration control in CMS, and the design could not be changed without a PR. Because a programmer's understanding of the requirements increases throughout the development phase, many changes are likely as the development process progresses. Tables 11 and 12 summarize the PR's issued during the development phase for the Mercury and the Pluto teams.

Deciding which PR data were of both interest and value in terms of assessing the development method was difficult. Inspections during the design and code processes were used to identify most of the obvious errors that occurred during the earlier stages of development. The original designs for both the Mercury and Pluto implementations had many errors. In fact, the modifications to each design were so extensive that a second round of inspections was conducted to review the entire design. Although the inspection of the initial versions of the source code did not find nearly as many problems as were found in the designs, a significant number of errors were identified here also. In general, the inspection process identified many problems that were representative of all types of errors in the design and code for both implementations.

In determining the data to present, we held to the notion that proper function of a software product is the fundamental measure of quality. Here, because the programmers were not allowed to execute the

Table 11. Summary of Mercury Development Activities

Development phase	Product	Verification activities	Related PR's
Design	Preliminary design	• Preliminary design review: 6 review sessions	PR's 1–13
		*Requirements change SDCR 2.3-2	PR 14
	Design	• Design review: 2 review sessions	PR's 15–22
		• Requirements change SDCR 2.3-4	PR 23
Code	Source code	• Programmer identified problems in design while developing code	PR 24
		• Code review: 2 review sessions	PR's 25–26
Integration	Executable object code	*Requirements change SDCR 2.3-6	PR 27
		• Requirements-based testing: Functional unit level Subframe level Frame level Trajectory level	PR 28
		• Structure-based testing	PR 29 (determined to not be problem)
		*Requirements change SDCR 2.3-7	PR 30

Table 12. Summary of Pluto Development Activities

Development phase	Product	Verification activities	Related PR's
Design	Preliminary design	• Preliminary design review: 9 review sessions	PR's 1–13
		*GCS specification mod SDCR 2.3-2	PR 14
	Design	• Design review: 2 review sessions	PR's 15–19
Code	Source code	• GCS specification mod SDCR 2.3-4	PR 20
		• Code review: 2 review sessions	PR's 21–23
Integration	Executable object code	*Requirements change SDCR 2.3-6	PR 27
		• Requirements-based testing: Functional unit level Subframe level Frame level Trajectory level	PR's 24 and 25 PR 26 PR 27
		• Structure-based testing	

code, the GCS implementations were executed for the first time during the integration phase. Testing started at the functional unit level and progressed through the trajectory level. In this section, the software errors identified during integration testing are discussed; then the results of the simulated operational testing are discussed.

What information about an error is of interest? Myers (ref. 41) suggests several questions to answer in the analysis of error data:

- What was done incorrectly?
- When was the error made?
- How was the error found?

In addition, were these errors related to any of the previous changes? For example, were some errors identified in the inspection activities not completely fixed? Or did some of the earlier fixes actually

introduce new errors? Table 13 provides the change information related to the integration testing of the Mercury implementation. Table 14 presents the change information from PR's 23–27 from the integration testing of the Pluto implementation. The error identification given in these tables consists of the PR number and the specific action item.

Note that errors 27-1 and 27-2 listed in table 14 were traced to errors in the GCS specification. The subsequent change to the GCS specification initiated PR 30 for the Mercury implementation. The design and code for both implementations were changed as a result of these specification errors.

Table 13. Error Data From Integration Phase for Mercury

Error ID	What was done incorrectly?	How was error found?	Phase error was introduced?	Relation to other changes?
28-1	Incorrect index was used (I instead of 1) in referencing variable in calculation.	Unit test	Code	None
28-2	Incorrect logic used in determining OPTIMAL_VELOCITY.	Unit test	Design	None

Table 14. Error Data From Integration Phase for Pluto

Error ID	What was done incorrectly?	How was error found?	Phase error was introduced?	Relation to other changes?
24-1	Wrong indices were used in calculating matrix entries (computational).	Unit test	Code	PR 23 addressed different error in same equations.
24-2.1	Wrong argument sent in call to subroutine (interface).	Unit test	Code	This error was originally identified in PR 23 in one location and was corrected. However, error occurred in three separate locations in source code.
24-2.2	Incorrect calculation of argument sent in call to subroutine (interface).	Unit test	Code	Error also was originally identified in PR 23 but was not completely corrected.
24-5	Lower and upper bounds were reversed in range check (other).	Unit test	Code	None
24-6	Integers were used when should have used real numbers (computational).	Unit test	Code	None
24-7	Used incorrect logic to implement table (logic).	Unit test	Code	None
24-8	Used incorrect precision for calculation (computational).	Unit test	Code	Error was identified in PR 23 but was not corrected.
24-9	Incorrect reference used in output message (other).	Reading code	Code	None
25-1	All necessary data not sent for cyclic redundancy check (other).	Unit test	Code	None
26-1	Typographical error in subroutine name.	Frame test	Code	None
26-2	Unnecessary continue statement.	Frame test	Code	None
27-1	Form of equation used for standard deviation could lead to negative square root.	Trajectory test	Requirements	Error initiated change in GCS specification (2.3-7.1). Equation was modified to eliminate possibility of negative square root. Both design and code were changed.
27-2	Valid data could lead to negative square root in equation.	Trajectory test	Requirements	Error initiated change in GCS specification (2.3-7.2). Equation was modified to eliminate possibility of negative square root. Both design and code were changed.

5.3. Summary of Findings From Operational Simulation

As mentioned above, a desirable feature of the GCS framework was the capability to run GCS versions in a back-to-back configuration for a large number of tests to simulate the operational performance of the software. To test this capability, the final versions of the Mercury and Pluto implementations were run in the simulator with the Venus prototype implementation. We chose to run the Venus implementation with the two DO-178B implementations of the GCS so that we would have an odd number of versions for comparison. Figure 9 illustrates the back-to-back testing of three GCS implementations.

Given that few errors were found in the integration tests for either of the two DO-178B implementations, we did not expect to find large numbers of errors in the back-to-back tests. Because the Venus implementation was developed by the system analyst and was used extensively in the development of the simulator, we did not expect to find many errors in Venus. However, we generally did expect to find some software errors.

A total of 7757 back-to-back tests (7757 complete trajectories) were run with the three GCS implementations. No differences were noted during those tests (i.e., for Boolean and integer comparisons, an exact match was noted; for real number comparisons, all values agreed within given tolerances). The vehicle successfully landed 97.1 percent of the time, which was within the successful landing rate of 95 percent that was established for this application. This result provides further confidence that the software was operating correctly (as expected). Some limit violations were noted; however, the limit violations were the same for all three implementations. This result could indicate that all three implementations had the same error; however, initial review of the limit violations indicated that some of the predefined limits needed to be adjusted.

Consequently, no errors were identified during the simulated operational testing. Thus, we have a good indication of the quality of the software products that have resulted from the process set forth in DO-178B. However, more testing is needed to estimate with a significant level of statistical confidence that no errors remain in the GCS implementations.

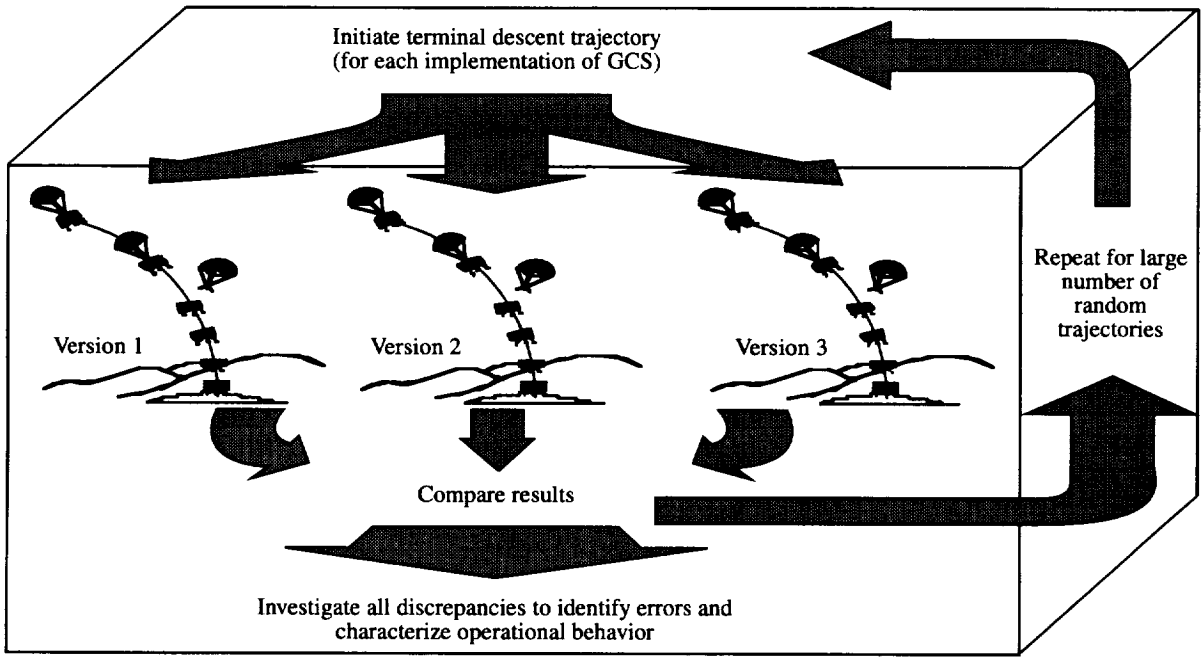


Figure 9. Back-to-back testing of multiple GCS implementations.

6. Project Summary

The GCS project had two primary goals: to establish a controlled environment for scientific experiments in software engineering and to conduct a case study to evaluate the effectiveness of the controlled environment. To accomplish these objectives, a framework that consisted of a requirements specification for a guidance and control application, a simulator for running the application, and a configuration management system to control the products of a software development process were produced. These framework elements allow researchers to conduct small-scale experiments in a consistent environment to scientifically evaluate hypotheses and assumptions regarding software engineering methods.

A case study, based on the DO-178B guidelines, was conducted to assess this framework and to gain experience in developing software to industry standards. The following subsections provide commentary on the experiment framework, software experimentation, and the DO-178B guidelines.

6.1. Comments on Framework

The elements of the framework (the GCS specification, the simulator, and the configuration management system) served their respective intended purposes. Table 15 shows the strengths and weaknesses that were noted. One of the biggest advantages to the experiment framework was the GCS specification. Few commercial software developers are willing to release requirement specifications for their products (for obvious reasons). The GCS specification provides a reasonable facsimile of a commercial avionics application developed with state-of-the-practice methods. Although the resulting source code is relatively small (approximately 2000 noncommented source lines), the GCS application is sufficiently complex to realize significant differences in the implementations.

The combination of the configuration management system and the GCS simulator provided the functionality necessary to control and simulate the operational performance of the GCS by implementing the back-to-back testing paradigm. The CMS provided capable management of the software versions. The automated configuration management system was invaluable, especially in working with multiple implementations; a complementary automated change reporting system would have been

Table 15. Strengths and Weaknesses of GCS Framework

Strengths	Weaknesses
• GCS specification provided reasonable basis for development of software (specification is expected to continue to improve with use). • CMS provided satisfactory management of software products. - Ability to easily retrieve intermediate versions of products during analysis of error data was helpful. • GCS simulator provided functionality necessary to simulate operational performance of GCS. - Performance could be enhanced by moving to different environment. • Elements in CMS library could be suitable for some level of experimentation. • Framework was available to public. • Framework did not impose constraints on design of experiments.	• Choice of VAX/VMS environment was regrettable because of declining interest in and support for this environment. - Framework could be ported to SUN/UNIX environment.[a] • Significant amount of resources may be required to develop complete GCS implementation starting from requirements (depending on development methods used). • Only one application was examined (guidance and control function), which was limited in size to approximately 2000 noncommented lines of source code. • Automated data collection system was not available.

[a]The GCS simulator and the CMS are both tied to the VAX/VMS environment. To move the framework to another platform, portions of the GCS simulator that are written in VMS command language would have to be rewritten. Other configuration management systems exist for other platforms.

helpful. The major drawback to the operational testing was that the simulator was implemented in a VAX/VMS environment, which has limited support. The performance of the simulator in running multiple implementations could have been improved by porting the simulator to a modern workstation environment; however, this change would not impact the GCS specification or the existing GCS implementations.

6.2. Comments on Software Experiments

The case study provided some general lessons in regard to conducting software experiments. The most significant lesson learned was that experimentation in software engineering is tedious, equivocal, and difficult to replicate, as is experimentation in the social sciences; gathering adequate and proper data for statistical analysis is extremely difficult. Our conclusions agree with those of Campbell and Stanley (ref. 32).

Software experimentation requires a clear goal that is stated as a measurable statistical hypothesis. An attitude of "we will develop some software using this method and see what we get" will not realize any meaningful results. Arthur and Nance (ref. 45) state that "the effectiveness of a [software development] methodology is defined as the degree to which a methodology produces a desired result." Perhaps this definition is true; however, a hypothesis such as "software developed in compliance with DO-178B will be highly reliable" is not acceptable for experimental research. The attribute "highly reliable" can have many different meanings, particularly when even the appropriate measure of software reliability is controversial. An experimental hypothesis must be stated in terms that are unambiguously quantifiable.

After the hypothesis is determined, plans for the experiment must be clearly defined; that is, rules should not be made up as the project progresses. This statement may seem obvious; however, if no mechanisms are in place to ensure that plans are complete prior to the start of the project, "planning as you go" becomes very easy. Unfortunately, a plan-as-you-go approach is not easy to replicate. Factors that affect the response of software products to the treatment under study (e.g., life-cycle model, hardware platform and tools, and programming language) should be clearly articulated to all participants at the start of the project. The development environment should be controlled to the extent possible to ensure that the difference in measurements is attributable only to the difference in treatments.

The experience level of the project participants is another significant factor in collecting adequate and relevant data. For example, although the participants in this case study were professional programmers, no one on the GCS project had any prior experience in developing software to any software standards, much less a rigorous standard like DO-178B. In fact, the DO-178B standard was in draft form when it was selected for use here. A great deal of project resources were expended in becoming familiar with DO-178B and the software engineering methods that comply with this standard.

Because the constructs of software are the result of human reasoning, software experiments cannot ignore the human factors that exist. Experience, intelligence, job security, personality, and other factors can impact a software development effort. In general, the level of control required to ensure that the differences noted in a software experiment are attributable only to variations in treatments (i.e., variations in applying different software engineering methods) is nearly impossible to achieve. The number of software developers that would be needed so that randomization techniques could be used to eliminate bias introduced through human factors would be huge. Consequently, the cost of conducting a full-scale randomized experiment would be prohibitive. The nature of software engineering may be such that it is not conducive to true scientific experimentation.

6.3. Comments on DO-178B

Lessons were also learned in regard to developing software in compliance with DO-178B. Table 16 lists negative and positive aspects of the DO-178B guidelines from the perspective of a first-time developer.

Although the DO-178B guidelines do not constrain a developer to a particular life-cycle model, the guidelines do require that specific objectives (given in annex A of DO-178B) be met. A well-defined life-cycle model is needed to meet these objectives. Section 11 of DO-178B lists the life-cycle data that should be generated during development. Nonetheless, the lack of templates, examples, or suggestions in the DO-178B guidelines makes it difficult to identify the type of evidence that may be required by the certificating authority to demonstrate that the objectives have been met. The following subsections contain observations regarding various life-cycle processes defined in DO-178B.

6.3.1. Planning Process

The timely development of the planning documents (table 3) is an important factor in an orderly software development program; these documents ensure that the final software product is traceable, testable, maintainable, and understandable. Defining these plans at the outset eliminates the tendency to plan as you go and gives the certificating authority an early development process. However, the DO-178B guidelines do not require that documentation be produced in any particular order or within any specific time frame. A specific schedule for the documentation could be included in the PSAC; however, the level of detail required in the PSAC is not clearly delineated. Ultimately, the life-cycle data should provide a complete history of the life cycle sufficient for the certificating authority to trace the requirements through development and completion.

6.3.2. Development Processes

Although section 11 of the DO-178B guideline calls for standards for software requirements, design, and code, little guidance is provided in DO-178B on the content of these standards. For this case study, few constraints were imposed on the programmers in developing the design and code. The intent from the research perspective was to give the programmers as much freedom as possible in developing the implementation; however, this freedom conflicted with the verification efforts required to ensure compliance with DO-178B. As mentioned above, the verification and assurance analysts had to become familiar with the product so that they could effectively verify that the product performed its intended function. Constraints such as limits on the level of complexity, style of documentation, and use of well-known tools made this familiarization process easier.

Table 16. Strengths and Weaknesses of DO-178B Guidelines

Strengths	Weaknesses
• Development process is well-defined and orderly.	• Document is difficult to read and apply. - Lack of examples/suggestions creates difficulty for first-time developers. (However, first-time developers are not intended audience.)
• Constraints and standards (for requirements, design, and code) can impact verification effort.	• Compliance with guidelines requires an extensive effort over and above writing source code.
• Tools can be helpful. - Organization and tracking tools can significantly increase efficiency. - Automation can significantly speed up paper-dependent processes (e.g., problem reporting).	• Tools can be harmful. - Time and money are required to learn new tools; this time should be accounted for in planning process. - All participants involved with output of development tool must understand tool.
• Software products are of fairly high quality (i.e., errors have not been identified to date in either implementation).	• Importance of objectives in tables in annex A is not obvious. - Evidence required to demonstrate that objectives have been met is not clearly identified. - Point in life cycle at which verification that such evidence exists is not made clear.

6.3.3. Integral Processes

Verification, configuration management, and quality assurance are considered integral processes in the DO-178B definition of life-cycle processes. Much of the development cost for this project was attributed to the verification activities, especially the development of the test cases. The guidance provided in DO-178B on the verification process is not always consistent with the objectives given in annex A. For example, although guidance is available on requirements-based testing for normal and robustness test cases, the objectives in annex A do not address normal and robustness testing. Only coverage of high- and low-level requirements is addressed in the objectives. With respect to structure-based testing, the effort to meet the MC/DC requirement did not reveal any errors; thus, the effectiveness of the MC/DC criteria should be the focus of further investigation.

The DO-178B guidelines place heavy emphasis on the configuration management process. The case study participants did not have prior experience with configuration management and found the guidance in DO-178B to be highly effective. In contrast, the guidance regarding software quality assurance was weak; the issue of the independence of the software quality assurance function for Level A software was particularly vague.

6.3.4. Tools

The use of software development tools yielded both positive and negative results. In general, use of the automated tools such as Teamwork, CMS, and Mathematica increased productivity. The change-reporting process used for the case study was a paper-based process; electronic forms would have been much easier to manage. One significant lesson learned was that sufficient training is essential for all project members on both the use of the tools and the review of tool output. For example, the programmers in the case study were required to develop their implementation designs using Teamwork; thus, the programmers were given a tutorial on how to use Teamwork. However, verification of the design involved inspection of the Teamwork model, but some of the inspectors did not have any knowledge of Teamwork and initially encountered difficulty in reviewing the Teamwork models. All project members who come in contact with a tool or the output of a tool must be educated about the tool. The planning process should allow time for all project participants to become familiar with new tools that will be used on the project.

7. Concluding Remarks

"Our greatest need now, in terms of future progress rather than short-term coping with current software engineering projects, is not for new languages or tools to implement our inventions, but for more in-depth understanding of whether our inventions are effective and why or why not" (Leveson, N. G.: IEEE Comput. Oct. 1994). The software experiments that have recently been conducted at Langley Research Center have generated data to characterize the software failure process and to improve methods for producing reliable software. The Guidance and Control Software (GCS) project, in particular, has established a framework for effectively conducting small-scale experiments to study the application of different software development methods. Here, the GCS framework was used for a case study to investigate the effectiveness of development and verification techniques that comply with the Requirements and Technical Concepts for Aviation RTCA/DO-178B guidelines, *Software Considerations in Airborne Systems and Equipment Certification*. This case study also has identified weaknesses in the framework.

The primary purpose of the framework is to facilitate the convenient collection of software error data for making statistical inferences for characterizing the software failure process and for evaluating the effectiveness of software development techniques. The case study provided valuable insight into the process of developing software in compliance with standards such as DO-178B and some general insight into conducting software experiments. From a broader perspective, the case study reinforced the notion that software engineering experiments are expensive, difficult to replicate, and even difficult to

interpret. With respect to studying software engineering methods, the level of control required to ensure that the effects seen in an experiment are attributable only to a variation of treatments is nearly impossible (at least within the resources typically available for such studies).

Although the framework does not eliminate many of the costs incurred in conducting software experiments, this experiment framework does provide a platform for consistent comparison for a small application. The framework and products of the case study are available to other researchers. The Federal Aviation Administration (FAA) has used the documentation from the case study to conduct software certification training for FAA airworthiness and certification specialists and representatives from the avionics industry; this training is designed to address avionics software issues that arise from the application of the DO-178B guidelines.

Ultimately, quantitative measures are needed to confirm the correlation between software development methods and product quality. Further research is needed to identify appropriate measures of software quality, especially in terms of safety and reliability, and to develop empirical methods for validating these measures.

NASA Langley Research Center
Hampton, VA 23681-2199
February 10, 1998

Appendix

Problem Reporting and Corrective Action

When presenting software error data, exposing the mechanisms used to gather that data is often helpful. This appendix addresses the content and identification of PR's for the life-cycle data generated for the DO-178B case study. For the GCS project, two different change-reporting systems were developed for the life-cycle data. The life-cycle data were divided into three categories: development products, support documentation, and records and results.

The development products included

- Design description
- Source code
- Executable object code

The support documentation included

- *Plan for Software Aspects of Certification*
- *Software Development Plan*
- *Software Requirements Standards*
- *Software Design Standards*
- *Software Code Standards*
- *Software Accomplishment Summary*
- *Software Verification Plan*
- *Software Verification Cases and Procedures*
- *Software Quality Assurance Plan*
- *Software Configuration Management Plan*
- *Software Life-Cycle Environment Configuration Index*
- *Software Configuration Index*
- *Software Requirements Data*

Records and results included

- *Software Verification Results*
- *Software Quality Assurance Records*
- *Problem Reports*
- *Software Configuration Management Records*

A system of PR's and AR's was used to document changes in the development products, and SDCR's were used to document changes in the support documents. No formal system of change reporting was used for the records and results.

The PR and AR forms, shown in figures 10 and 11, respectively, were used to document problems and subsequent changes to the development products that occurred during the development of the GCS implementations. The PR contained

- Information regarding the point in the development process at which the problem was identified.
- The configuration identification of the artifact.
- A description of the problem (such as noncompliance with project standards or output deficiency).
- A history log for tracking the progress and resolution of the problem.

GCS Problem Report

page 1 of ____

1. PR #:	2. Planet:	3. Discovery Date:	4. Initiator & Role:

5. Activity at Discovery:

Development Phases \ Activity	Design Review	Code Review	Reading Code	Reading Specification	Test Readiness Review	Test Completion Review	Test Case Creation	Test Case Execution	Regression	Other
Design		■	■		■	■	■	■		
Code	■				■	■	■	■		
Unit Testing										
Functional	■	■								
Structural	■	■								
Subframe Testing	■	■								
Frame Testing	■	■								
Top-Level Simulator	■	■								
Integration Testing										

6. Description of Problem:

7. Artifact Identification:

☐ Design Description ☐ Support Documentation _____

☐ Source Code ☐ Other _____

☐ Executable Object Code _____

8. Test Case Identification:

9. History Log:

Date To	Date From	Person	Comments	AR#

10. Total # of Changes: ☐ 11. Total # of No Changes: ☐

12. Initiator Signature & Date	13. SQA Signature & Date

Figure 10. GCS PR form.

GCS Action Report

page 1 of ____

1. AR #:	2. Planet:	3. Date of Action:	4. Respondent & Role:

5. Artifact Identification:

☐ Design Description ☐ Support Documentation _____

☐ Source Code ☐ Other _____

☐ Executable Object Code _____

6. Description of Action:

7. Was this action related to another action(s)?

☐ Yes AR#(s) _____

☐ No

☐ I don't know

Figure 11. GCS AR Form.

All problems were investigated to determine if a true error had been detected; if so, corrective action was taken and properly documented. Each identified error was traced to determine at which point the error was introduced. The AR form was used to capture relevant information about the action that was taken in response to an issued PR. The AR contained the configuration identification of the artifact that was affected and a description of the change that was made to the artifact in response to the PR. Change control procedures, as described in the *Software Configuration Management Plan*, were followed when an actual change was made to a configuration item. In cases for which no change was required in response to the PR, the AR form contained the justification for not making any changes.

Problem Reporting for Support Documentation

The problem and change reporting for the support documentation was accomplished through the use of SDCR's. Although the SDCR shown in figure 12 did not capture as much detailed information as the PR, this form did capture the information necessary to comply with paragraph 7.2.3 of DO-178B. Once a support document entered the configuration management system, further changes to that document were controlled through the SDCR forms; that is, all changes to support documentation were required to be accompanied by an approved SDCR. Each configuration item that was part of the support documentation had its own set of change reports. The software quality assurance representative was required to keep a log of all change reports for each configuration item.

Numbering System for PR's and AR's

Separate sets of PR's and AR's were maintained for the development products for each GCS implementation. The identification numbers for the PR's and AR's were of the form *a.b*, where *a* is the chronological number of the PR and *b* is the chronological number of the action taken in response to PR *a*. Thus, the PR's were numbered 1.0, 2.0, 3.0, and so on. Successive actions (noted on individual AR forms) in response to a given PR were numbered <PR#>.1, <PR#>.2, <PR#>.3, and so on.

For example, consider the third problem identified for a given implementation. The PR number would be 3.0. Now suppose that two actions are taken in response to this PR. The AR numbers would be 3.1 and 3.2.

Support Documentation Change Report

page 1 of ____

1. Configuration Item:	2. Date:	3. Modification #:
4. Part of Configuration Item Affected:		
5. Reason for Modification:		
6. Modification:		
7. SQA Signature & Date:		

Figure 12. SDCR form.

References

1. Sweet, William, ed.: The Glass Cockpit—Interfacing With the Pilot. *IEEE Spectrum*, Sept. 1995, pp. 30–38.

2. Mellor, Peter: 10 to the -9 and All That: The Non-Certification of Flight-Critical Software. Paper presented at ESCOM '96 (Wilmslow, Cheshire), 1996.

3. Airworthiness Directives. 14 CFR Part 39, Amendment 39-9494, FAA, Feb. 1996. (Available from DTIC as AD 96-02-06.)

4. Joch, Alan: How Software Doesn't Work. *BYTE*, Dec. 1995, pp. 49–60.

5. Gibbs, W. Wayt: Software's Chronic Crisis. *Sci. American*, Sept. 1994, pp. 86–95.

6. Peterson, Ivars: *Fatal Defect—Chasing Killer Computer Bugs*. Random House, Inc. 1995.

7. Butler, Ricky W.; and Finelli, George B.: The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. *IEEE Trans. Softw. Eng.*, vol. 19, no. 1, Jan. 1993, pp. 3–12.

8. Finelli, George B.: NASA Software Failure Characterization Experiments. *Reliab. Eng. & Syst. Saf.*, vol. 32, 1991, pp. 155–169.

9. Leveson, Nancy G.: High-Pressure Steam Engines and Computer Software. *IEEE Comput.*, Oct. 1994, pp. 65–73.

10. Brooks, Frederick P., Jr.: No Silver Bullet—Essence and Accidents of Software Engineering. *IEEE Comput.*, Apr. 1987, pp. 10–19.

11. Williams, Tom, ed.: It Takes More Than a Keen Nose to Track Down Software Bugs. *Comput. Design*, Sept. 1993, pp. 67–70 and 90–91.

12. Leveson, Nancy G.: *SAFEWARE—System Safety and Computers*. Addison-Wesley Publ. Co., 1995.

13. Wiener, Lauren Ruth: Digital Woes—Why We Should Not Depend on Software. Addison-Wesley Publ. Co., 1993.

14. Neumann, Peter G.: *Computer Related Risks*. ACM Press, 1994.

15. Holloway, C. Michael: Software Engineering and Epistemology. ACM *SIGSOFT Softw. Eng. Notes*, vol. 20, no. 2, Apr. 1995, pp. 20–21.

16. Eckhardt, Dave E.; Caglayan, Alper K.; Knight, John C.; Lee, Larry D.; McAllister, David F.; Vouk, Mladen A.; and Kelly, John P. J.: An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability. *IEEE Trans. Softw. Eng.*, vol. 17, no. 7, July 1991, pp. 692–702.

17. Knight, John C.; and Leveson, Nancy G.: An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 1, Jan. 1986, pp. 96–109.

18. Brilliant, Susan S.; Knight, John C.; and Leveson, Nancy G.: Analysis of Faults in an *N*-Version Software Experiment. *IEEE Trans. Softw. Eng.*, vol. 16, no. 2, Feb. 1990, pp. 238–247.

19. Leveson, Nancy G.; Cha, Stephen S.; Knight, John C.; and Shimeall, Timothy J.: The Use of Self Checks and Voting in Software Error Detection: An Empirical Study. *IEEE Trans. Softw. Eng.*, vol. 16, no. 4, Apr. 1990, pp. 432–443.

20. Shimeall, Timothy J.; and Leveson, Nancy G.: An Empirical Comparison of Software Fault Tolerance and Fault Elimination. *IEEE Trans. Softw. Eng.*, vol. 17, no. 2, Feb. 1991, pp. 173–182.

21. Hecht, H.; Sturm, W. A.; and Trattner, S.: *Reliability Measurement During Software Development*. NASA CR-145205, 1977.

22. Hecht, H.: *Measurement Estimation and Prediction of Software Reliability*. NASA CR-145135, 1977.

23. Maxwell, F. D.: *The Determination of Measures of Software Reliability*. NASA CR-158960, 1978.

24. Nagel, Phyllis M.; and Skrivan, James A.: *Software Reliability: Repetitive Run Experimentation and Modeling*. NASA CR-165836, 1982.

25. Nagel, P. M.; Scholz, F. W.; and Skrivan, J. A.: *Software Reliability: Additional Investigation Into Modeling With Replicated Experiments*. NASA CR-172378, 1984.

26. Dunham, Janet R.: Experiments in Software Reliability: Life-Critical Applications. *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 1, Jan. 1986, pp. 110–123.

27. Dunham, J. R.; and Lauterbach, L. A.: *An Experiment in Software Reliability Additional Analyses Using Data From Automated Replications*. NASA CR-178395, 1987.

28. Dunham, Janet R.; and Pierce, John L.: *An Empirical Study of Flight Control Software Reliability.* NASA CR-178058, 1986.

29. Software Considerations in Airborne Systems and Equipment Certification. Doc. No. RTCA/DO-178B, RTCA, Inc., Dec. 1, 1992.

30. Hamlet, Dick: Predicting Dependability by Testing. *Software Engineering Notes—Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, vol. 21, no. 3, May 1996, pp. 84–91.

31. Fenton, Norman; Pfleeger, Shari Lawrence; and Glass, Robert L.: Science and Substance: A Challenge to Software Engineers. *IEEE Softw.*, vol. 11, July 1994, pp. 88–95.

32. Campbell, Donald T.; and Stanley, Julian C.: *Experimental and Quasi-Experimental Designs for Research.* Houghton Mifflin Co., 1963.

33. Holmberg, Neil A.; Faust, Robert P.; and Holt, H. Milton: *Viking '75 Spacecraft Design and Test Summary. Volume I—Lander Design.* NASA RP-1027, 1980.

34. Hatley, Derek J.; and Pirbhai, Imtiaz A.: *Strategies for Real-Time System Specification.* Dorset House Publ. Co., Inc., 1987.

35. Team*work/SA*®, Team*work/RT*®*—User's Guide.* Cadre Technol. Inc., 1990.

36. *Guide to VAX DEC/Code Management System.* Digital Equip. Corp., 1989.

37. Basili, Victor R.; Selby, Richard W.; and Hutchens, David H.: Experimentation in Software Engineering. *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 7, July 1986, pp. 733–743.

38. Pfleeger, Shari Lawrence: Experimental Design and Analysis in Software Engineering. *ACM SIGSOFT Softw. Eng. Notes.*
Part 1: The Language of Case Studies and Formal Experiments, vol. 19, no. 4, Oct. 1994, pp. 16–20.
Part 2: How to Set Up an Experiment. vol. 20, no. 1, Jan. 1995, pp. 22–26.
Part 3: Types of Experimental Design, vol. 20, no. 2, Apr. 1995, pp. 14–16.
Part 5: Analyzing the Data, vol. 20, no. 5, Dec. 1995, pp. 14–17.

39. RTCA, Inc., Document RTCA/DO-178B. AC No. 20-115B, FAA, Jan. 11, 1993.

40. Fagan, M. E.: Design and Code Inspections to Reduce Errors in Program Development. *IBM Syst. J.*, vol. 15, no. 3, 1976, pp. 182–211.

41. Myers, Glenford J.: *The Art of Software Testing.* John Wiley & Sons, Inc., 1979.

42. Wolfram, Stephen: *Mathematica—A System for Doing Mathematics by Computer.* Addison-Wesley Publ. Co., Inc., 1988.

43. *Analysis of Complexity Tool*™*—User's Instructions.* T. J. McCabe & Assoc., Inc., 1992.

44. Chilenski, John Joseph; and Miller, Steven P.: Applicability of Modified Condition/Decision Coverage to Software Testing. *Softw. Eng. J.*, vol. 9, no. 5, Sept. 1994, pp. 193–200.

45. Arthur, James D.; and Nance, Richard E.: A Framework for Assessing the Adequacy and Effectiveness of Software Development Methodologies. *Proceedings of the 15th Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Nov. 1990.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 07704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1998	Technical Memorandum

4. TITLE AND SUBTITLE
Framework for Small-Scale Experiments in Software Engineering

Guidance and Control Software Project: Software Engineering Case Study

5. FUNDING NUMBERS
WU 519-30-31-01

6. AUTHOR(S)
Kelly J. Hayhurst

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-17621

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA/TM-1998-207666

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited
Subject Category 38 Distribution: Standard
Availability: NASA CASI (301) 621-0390

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

Software is becoming increasingly significant in today's critical avionics systems. To achieve safe, reliable software, government regulatory agencies such as the Federal Aviation Administration (FAA) and the Department of Defense mandate the use of certain software development methods. However, little scientific evidence exists to show a correlation between software development methods and product quality. Given this lack of evidence, a series of experiments has been conducted to understand why and how software fails. The Guidance and Control Software (GCS) project is the latest in this series. The GCS project is a case study of the Requirements and Technical Concepts for Aviation RTCA/DO-178B guidelines, *Software Considerations in Airborne Systems and Equipment Certification*. All civil transport airframe and equipment vendors are expected to comply with these guidelines in building systems to be certified by the FAA for use in commercial aircraft. For the case study, two implementations of a guidance and control application were developed to comply with the DO-178B guidelines for Level A (critical) software. The development included the requirements, design, coding, verification, configuration management, and quality assurance processes. This paper discusses the details of the GCS project and presents the results of the case study.

14. SUBJECT TERMS
Software engineering; Software experiment; Software standards; DO-178B guidelines

15. NUMBER OF PAGES
46

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102